A $^{+}$CAL User's Manual

Leslie Lamport

27 Dec 2005

Contents

Preface **1**

1 Introduction **2**

2 Getting Started **4**
2.1 Typing the Algorithm 4
2.2 The TLA$^+$ Module 6
2.3 Translating and Executing the Algorithm 7
2.4 Checking the Results 9
2.5 Checking Termination 9
2.6 A Multiprocess Algorithm 10

3 The Language **16**
3.1 Expressions 17
3.2 The Statements 17
3.2.1 Assignment 17
3.2.2 If 18
3.2.3 Either 19
3.2.4 While 20
3.2.5 When 20
3.2.6 With 20
3.2.7 Skip 21
3.2.8 Print 21
3.2.9 Assert 21
3.2.10 Call and Return 22
3.2.11 Goto 22
3.3 Processes 22
3.4 Procedures 23
3.5 Macros 25
3.6 Definitions 26
3.7 Labels 26
3.8 The Translation's Definitions and Declarations 27

4 Checking the Algorithm **30**
4.1 Running the Translator 30
4.2 Specifying the Constants 31
4.3 Constraints 33
4.4 Understanding TLC's Output 33

4.5 Invariance Checking 35
4.6 Termination and Liveness 35
4.7 Additional TLC Features 38
4.7.1 Deadlock Checking 38
4.7.2 Multithreading 39
4.7.3 Symmetry 39

5 TLA$^+$ Expressions and Definitions 41
5.1 Numbers 41
5.2 Strings 41
5.3 Boolean Operators 42
5.4 Sets 43
5.5 Functions 45
5.6 Records 46
5.7 The Except Construct 46
5.8 Tuples and Sequences 47
5.9 Miscellaneous Constructs 48
5.10 Temporal Operators 49
5.10.1 Fairness 49
5.10.2 Liveness 51
5.10.3 One Algorithm Implementing Another 52
5.11 TLA$^+$ Definitions 52

References 53

A The Grammar 54

B The TLA$^+$ Translation 57
B.1 The *FastMutex* Algorithm 57
B.2 Procedures 61

Index 66

Preface

This is an instruction manual for the +CAL algorithm language. Section 1 explains what an algorithm language is and why you'd want to use one. Section 2 tells you what you need to know to get started using +CAL. After reading it, you'll be able to write and check +CAL algorithms.

You can read the other parts of this manual as you need them. The table of contents and the index can help you find what you need. Pages 63–65 at the end, just before the index, contain a series of tables that summarize a lot of useful information. The rest of the manual is arranged in the order you're likely to want to look at it:

- Section 3 describes the things you'll find in most programming language manuals, like the statements of the language. Once you've started writing +CAL algorithms, you should browse this chapter to learn about features of +CAL not mentioned in Section 2.

- We run programs, but we check algorithms. Section 2 gets you started using the translator and TLC model checker to check +CAL algorithms. Section 4 tells you more about the translator and TLC. It's mostly about TLC, describing some of its additional features and how to use it to debug an algorithm. You should go to Section 4 if you don't understand what the translator or TLC is trying to say when it reports an error.

- Section 5 is mainly about writing +CAL expressions. The expression language of +CAL is much richer and more powerful than that of any programming language because it is based on mathematics, not on programming. The ten or so pages about expressions in Section 5 just introduce the subject. You can learn more from the book *Specifying Systems*, referred to here as the TLA$^+$ book [2], or from any books on the elementary mathematics of sets, functions, and logic—especially ones written by mathematicians and not computer scientists.

- Section A of the appendix contains a BNF grammar of +CAL. The subject of Appendix Section B will make no sense to you until you've read Section 1.

I wish to thank the people who helped make +CAL possible. Keith Marzullo collaborated in the writing of the translator and helped with the design of the +CAL language. Georges Gonthier made many useful suggestions for the language.

1 Introduction

+CAL (pronounced "plus-CAL") is an algorithm language. An algorithm language is meant for writing algorithms, not programs. Algorithms differ from programs in several ways:

- Algorithms perform operations on arbitrary mathematical objects, such as graphs and vector spaces. Programs perform operations on simple objects such as Booleans and integers; operations on more complex data types must be coded using lower-level operations such as integer addition and method invocation.

- A program describes one method of computing a result; an algorithm may describe a class of possible computations. For example, an algorithm might simply require that a certain operation be performed for all values of i from 1 to N. A program specifies in which order those operations are performed.

- Execution of an algorithm consists of a sequence of steps. An algorithm's computational complexity is the number of steps it takes to compute the result; defining a concurrent algorithm requires specifying what constitutes a single (atomic) step. There is no well-defined notion of a step of a program.

These differences between algorithms and programs are reflected in the following differences between +CAL and programming languages.

- The language of +CAL expressions is TLA$^+$, a high-level specification language based on set theory and first-order logic [2]. TLA$^+$ is infinitely more expressive than the expression language of any programming language. Even the subset of TLA$^+$ that can be executed by the TLC model checker is far more expressive than any programming language.[1]

- +CAL provides simple constructs for expressing nondeterminism.

- +CAL uses labels to describe the algorithm's steps. This works quite well for describing the grain of atomicity in the absence of procedure calls. Invoking a procedure and returning from a procedure must begin

[1] SETL [3] provides many of the set-theoretic primitives of TLA$^+$, but it can implement higher-level operations only by programming them with procedures and it cannot conveniently express nondeterminism.

new steps, which does limit the flexibility of describing atomicity in algorithms with procedures. However, +CAL's macro facility and the ability to define operators in TLA+ and use them in +CAL expressions often make procedure calls unnecessary.

The primary goals of a programming language are efficiency of execution and ease of writing large programs. The primary goals of an algorithm language are making algorithms easier to understand and helping to check their correctness. Efficiency matters when executing a program that implements the algorithm. Algorithms are much shorter than programs, typically dozens of lines rather than thousands. An algorithm language doesn't need complicated concepts like objects or sophisticated type systems that were developed for writing large programs.

It is easy to write a +CAL algorithm that cannot be executed—for example, one containing a statement that assigns to x the smallest integer for which Goldbach's conjecture[2] is false, if one exists, or else the value 0. An unexecutable algorithm can be interesting, and may represent a step in the development of a practical algorithm. However, most +CAL users will want to execute their algorithms. The +CAL translator compiles a +CAL algorithm into a TLA+ specification. If the algorithm manipulates only finite objects in a sensible way, then the TLC model checker will probably be able to execute that specification. When used in model-checking mode, TLC will check all possible executions of the algorithm. It can also be used in simulation mode to check randomly generated executions.

[2]Goldbach's conjecture, which has not been proved or disproved, asserts that any even number greater than 2 is the sum of two primes.

2 Getting Started

I assume here that you've programmed in an imperative language like Java or Pascal or C. I will therefore not bother to explain the meaning of something like a `while` statement that appears in such languages. You can find the meaning of `while` and all other +CAL statements in Section 3. (The index can help you.)

2.1 Typing the Algorithm

As an example, consider the following bit of +CAL code that describes Euclid's algorithm, adapted from a version given by Sedgewick [4, page 8]. It sets `v` to the gcd (greatest common divisor) of `u` and `v`.

```
lp: while u ≠ 0 do   (* ≠ is typed # or /= or \neq . *)
      if u < v then u := v || v := u ;  \* swap u and v.
      end if ;
    a: u := u - v;
    end while ;
```

Comments indicate how to type symbols such as "≠" that appear in the examples. A complete list of the ASCII versions of symbols appears in Table 5 on page 65.

You should find this code easy to understand, except for the "`||`" in the `then` clause on the second line. Assignment statements separated by "`||`"s (rather than by semicolons) are executed simultaneously by first evaluating all the right-hand sides, then doing the assignments. Thus, as the comment says, that multiple assignment swaps the values of `u` and `v`.

Also unusual in this piece of code is the presence of the labels `lp` and `a`. They serve to delimit the steps of the algorithm. A step consists of the execution from one label to the next. One iteration of the `while` loop, when `u` is nonzero, consists of two steps:

- The step from `lp` to `a`, which executes the test `u` ≠ `0` and the `if` statement.
- The step from `a` to `lp`, which executes the assignment statement labeled by `a`.

If `u` equals zero, then the step starting at `lp` executes the `while` test and continues until the next label following the `while` statement. An implicit

label `Done` is assumed to follow the last statement of the algorithm. The first statement of the algorithm, right after the `begin`, must be labeled. The rules for what other statements must and must not have labels are given with the descriptions of the statements in Section 3; the rules are summarized in Section 3.7 on page 26.

The snippet of algorithm also indicates the two ways comments are written: either begun with "`\*`" and ended by the end of the line, or enclosed in matching "`(*`" and "`*)`" delimiters. Comments can be nested, so you can use "`(*`" and "`*)`" to comment out commented code.

Let's now put this piece of code into a complete algorithm. The algorithm begins

```
--algorithm EuclidAlg
```

where we've given it the name `EuclidAlg`. We next declare the variables `u` and `v` and specify their initial values. (We could omit their initial values and initialize them with assignment statements, but it's better to do it this way.) Just to illustrate the two kinds of initialization, we give `u` the initial value 24, but let the initial value of `v` be any integer from 1 through some parameter `N`.

```
variables u = 24; v ∈ 1..N;
```
`\*` `∈` is typed `\in` .

The declaration of `v` asserts that its initial value is an element of the set `1 .. N` of integers from 1 through `N`.

We add `print` statements to print out the initial values of the variables and the final value of `v`. The `print` statement can print the value of any arbitrary expression; to print multiple values, we can either let that expression be a tuple or else use multiple `print` statements. The complete algorithm is as follows, where the `while` loop is the same as above.

```
--algorithm EuclidAlg
  variables u = 24 ; v ∈ 1 .. N ;
  begin  a: print ⟨u, v⟩ ;
         lp: while u ≠ 0 do
                   ⋮
             end while ;
             print ⟨"have gcd", v⟩ ;
  end algorithm
```
`\*` `⟨ ... ⟩` is typed `<< ... >>` .

2.2 The TLA$^+$ Module

The translated version of the algorithm is put inside a TLA$^+$ module. The algorithm must go in the same file as the module. The module begins

MODULE *Euclid*

which is typed as

```
---------------- MODULE Euclid ----------------
```

(The number of dashes in each "`----` $\cdots$ `----`" doesn't matter, as long as there are at least four.) The module name is arbitrary, but a module named *Euclid* must go in a file named `Euclid.tla`.

The module next imports two standard TLA$^+$ modules.

EXTENDS *Naturals*, *TLC*

The *Naturals* module defines common operators on natural numbers, including subtraction ("$-$") and the operator "`..`" that appear in the algorithm's expressions. The *TLC* module is needed if the algorithm uses a `print` statement. The EXTENDS statement must be the first statement in the module.

Next, the module declares the parameter `N`.

CONSTANT N

Every symbol or identifier that appears in an expression in the algorithm must be either (a) a built-in TLA$^+$ operator like $=$ or $\langle \ldots \rangle$, (b) declared or defined in the module, or (c) declared or defined in an imported module.

The module must also contain two single-line comments:

```
\*  BEGIN TRANSLATION
\*  END TRANSLATION
```

The translator will delete everything between those two lines and replace it with the TLA$^+$ translation of the algorithm. The module ends with

which is typed as a string of four or more "`=`" characters.

The algorithm itself may appear in the module within a single comment:

```
(*  --algorithm EuclidAlg
          .
          .
          .
    end algorithm   *)
```

It may also be placed before or after the module in file `Euclid.tla`. However, it is best to put it before the "`BEGIN TRANSLATION`" line; otherwise, the line numbers reported in error messages may be wrong.

2.3 Translating and Executing the Algorithm

The translator is a Java program. We run it to translate the algorithm by typing

```
java pcal.trans Euclid
```

to a Windows or Unix/Linux command-line window. After translating the algorithm, we can execute it by running the TLC model checker. The translator creates a *configuration file* named `Euclid.cfg`. We must add to that file a statement that assigns values to the parameters. We assign the value 3000 to the parameter N by putting the statement

```
CONSTANT N = 3000
```

in the configuration file. A comment in the file indicates where this statement should go. (The configuration file may contain comments begun by `\*` and ended by the end of the line.) We can now run TLC with the command

```
java tlc.TLC -simulate -depth 200 Euclid
```

This runs TLC in *simulation* mode, in which it repeatedly chooses an arbitrary initial value of `v` in the set `1..3000` and executes the algorithm for at most 200 steps. (If the `-depth` option is omitted from the command line, the default value of 100 steps is used.) TLC produces a few lines of output in which its says what it's doing when processing the input file, followed by a gush of output like

```
<< 24, 1005 >>
<< "have gcd", 3 >>
<< 24, 200 >>
<< "have gcd", 8 >>
<< 24, 2717 >>
<< "have gcd", 1 >>
<< 24, 898 >>
<< "have gcd", 2 >>
<< 24, 1809 >>
      ⋮
```

that ends only when we stop the TLC program (usually by typing a control-C character).

Instead of having TLC randomly generate possible executions, we can run it in *model-checking mode*, in which it checks all possible executions of the algorithm. To avoid a huge mass of output, let's change the configuration file to have it set N to 4, so there are only 4 possible executions of the algorithm. We run TLC in model-checking mode by typing

```
java tlc.TLC Euclid
```

and it produces the following output:

```
<< 24, 1 >>
<< 24, 2 >>
<< 24, 3 >>
<< 24, 4 >>
<< "have gcd", 4 >>
<< "have gcd", 3 >>
<< "have gcd", 2 >>
<< "have gcd", 1 >>
```

TLC has checked the four possible executions, producing the eight possible executions of the `print` statements. But it did not perform those executions separately. Instead, TLC found all reachable states using a breadth-first search. In doing so, it performed the four possible first steps before performing any of the four possible last steps.

If you want sensible output from running TLC in model-checking mode, you should have the algorithm execute only a single `print` statement at the end. For our example algorithm, this requires saving the initial value of `v` in a separate variable. So, we modify the algorithm by introducing a new variable `v_ini` whose initial value is the initial value of `v`.

```
--algorithm EuclidAlg
  variables u = 24 ; v ∈ 1 .. N ; v_ini = v ;
  begin  lp: while u ≠ 0 do ...
             end while ;
             print <<24, v_ini, "have gcd", v>> ;
  end algorithm
```

Translating and running TLC in model-checking mode on this algorithm produces the output

```
<< 24, 4, "have gcd", 4 >>
<< 24, 3, "have gcd", 3 >>
<< 24, 2, "have gcd", 2 >>
<< 24, 1, "have gcd", 1 >>
```

2.4 Checking the Results

We don't have to print the results and examine them by hand to check them. We can let TLC do the checking by using an `assert` statement. Suppose we have defined $gcd(x, y)$ to be the gcd of x and y. We can then replace the `print` statement in algorithm *EuclidAlg* by

```
assert v = gcd(24, v_ini)
```

TLC will print an error message if this statement is executed when `v` does not equal `gcd(24, v_ini)`. For this to work, the operator *gcd* must be defined in the TLA$^+$ module, before the translated algorithm—that is, before the "`BEGIN TRANSLATION`" line. You may be able to understand the TLA$^+$ definition of *gcd* knowing that:

- $gcd(x, y)$ is defined to be the largest integer that divides both x and y.
- An integer p divides an integer q iff (if and only if) $q \,\%\, p$ equals 0, where $q \,\%\, p$ is the remainder when q is divided by p.
- The gcd of x and y is at most equal to x (or y).

The standard TLA$^+$ operators that are used in the definition are briefly explained in Tables 1 and 2 on pages 63 and 64. Here is the definition; give it a try.

$$
\begin{array}{l}
gcd(x,\, y) \triangleq \text{CHOOSE } i \in 1 \,..\, x : \\
\qquad \land\, x \,\%\, i = 0 \\
\qquad \land\, y \,\%\, i = 0 \\
\qquad \land\, \forall\, j \in 1 \,..\, x : \land\, x \,\%\, j = 0 \\
\qquad\qquad\qquad\qquad\quad \land\, y \,\%\, j = 0 \\
\qquad\qquad\qquad\qquad\quad \Rightarrow i \geq j
\end{array}
$$

If you can't understand it now, you should be able to after reading Section 5.

2.5 Checking Termination

To check that algorithm `EuclidAlg` always terminates, we perform the translation with the command

```
java pcal.trans -translation Euclid
```

This produces the appropriate translation and configuration file to instruct TLC to check for termination. If TLC discovers a non-terminating execution, it will print an error message indicating that property *Termination* is

violated and will describe the non-terminating trace. Section 4.4 on page 33 explains how to interpret TLC's error messages.

You can check an algorithm for termination only if every variable is initialized with a value of the proper type. Here, "variable" means every TLA$^+$ variable declared by the translation. As explained in Section 3.8, these variables include procedure parameters as well as the algorithm's global variables and local procedure variables. Procedures are described in Section 3.4, and page 35 of Section 4.5 explains how to assign initial values to procedure parameters. If a variable is not initialized, termination checking will cause TLC to produce a mysterious error message of the form:

```
Error:  Attempted to check equality of ...  with ...  {}
```

2.6 A Multiprocess Algorithm

Algorithm `EuclidAlg` is a uniprocess algorithm, with only a single thread of control. We now look at an example of a multiprocess algorithm written in $^+$CAL. The example is the Fast Mutual Exclusion Algorithm [1]. The algorithm has N processes, numbered from 1 through N. Figure 1 on the next page is the original description of process number i, except with the noncritical section and the outer infinite loop made explicit. Angle brackets enclose atomic operations (steps). For example, the evaluation of the expression $y \neq 0$ in the first **if** statement is performed as a single step. If that expression equals *true*, the next step of the process sets $b[i]$ to *false*. The process's next atomic operation is the execution of the **await** statement, which is performed only when y equals 0. (The step cannot be performed when y is not equal to 0.)

The $^+$CAL version of this algorithm is in Figure 2 on page 12. After the algorithm name comes the declaration of the global variables:

```
variables x = 0 ; y = 0 ; b = [i ∈ 1..N ↦ FALSE] ;
```

The declaration of `b` states that it is initially an array indexed by the set `1..N` such that `b[i]` equals `FALSE` for every `i` in `1..N`. (The symbol "↦" is typed "`|->`".) The expression

$$[v \in S \mapsto v + 1]$$

equals an array A indexed by the set S such that $A[v] = v + 1$ for every v in S. What programmers call an array, mathematicians call a function. Like a mathematician, I usually call A a *function with domain* S rather than an *array indexed by* S. However, TLA$^+$ and $^+$CAL use programmers' square

```
 ncs: noncritical section;
start: ⟨b[i] := true⟩;
       ⟨x := i⟩;
       if ⟨y ≠ 0⟩ then ⟨b[i] := false⟩;
                       await ⟨y = 0⟩;
                       goto start   fi;
       ⟨y := i⟩;
       if ⟨x ≠ i⟩ then ⟨b[i] := false⟩;
                       for j := 1 to N do await ⟨¬b[j]⟩ od;
                       if ⟨y ≠ i⟩ then await ⟨y = 0⟩;
                                       goto start   fi   fi;
       critical section;
       ⟨y := 0⟩;
       ⟨b[i] := false⟩;
       goto ncs
```

Figure 1: Process i of the fast mutual exclusion algorithm, based on the original description.

brackets instead of mathematicians' parentheses to represent array/function application. The $\mapsto$ construct and other TLA$^+$ notation for functions is explained in Section 5.5 on page 45.

The algorithm continues with

```
process Proc ∈ 1..N
```

This statement begins a collection named `Proc` of N processes, each process identified by a number in `1..N`. The statement

```
variable j ;
```

declares `j` to be a local variable of these processes, meaning that each of the N processes has its own separate variable `j`. A local or global variable `z` can be initialized by a declaration of the form

`variable z =` *exp* `;` or `variable z ∈` *exp* `;`

The matching "`begin`" and "`end process`" enclose the code for each process in the collection `Proc`, where `self` is the identifier of that process (in this example, a number in `1..N`). The correspondence between the original pseudo-code in Figure 1 and the code for each process `self` of the $^+$CAL algorithm in Figure 2 should be clear. We represent the noncritical and

```
--algorithm FastMutex
variables x = 0 ; y = 0 ; b = [i ∈ 1..N ↦ FALSE] ;

process Proc ∈ 1..N
  variable j ;
  begin
    ncs: while TRUE do
              skip ;  \* The noncritical section.
       start: b[self] := TRUE ;
          l1: x := self ;
          l2: if y ≠ 0 then l3: b[self] := FALSE ;
                            l4: when y = 0 ;
                                goto start ;
              end if ;
          l5: y := self ;
          l6: if x ≠ self
                then l7: b[self] := FALSE ;
                         j := 1 ;
                      l8: while j ≤ N do when ~b[j] ;
                                        j := j+1 ;
                         end while ;
                      l9: if y ≠ self then l10: when y = 0 ;
                                                goto start ;
                         end if ;
               end if;
           cs: skip ;  \* The critical section.
         l11: y := 0 ;
         l12: b[self] := FALSE ;
         end while ;
end process

end algorithm
```

Figure 2: The fast mutual exclusion algorithm in +CAL.

critical sections as atomic `skip` operations whose execution consists of a single "no-op" step that does nothing. The **await** statement of the original version is represented by the $^+$CAL `when` statement. A step containing a statement "`when` *exp*" can be executed only if the expression *exp* equals `TRUE`. Think of the execution of the step as aborting, and having no effect, if *exp* equals `FALSE`.

The original algorithm uses a **for** loop to test the values of $b[j]$ in increasing order of j. The `for` loop is represented in the $^+$CAL version by the `while` loop at label `l8` and the assignment statement that precedes it. However, the algorithm works if the $b[j]$ are tested in arbitrary order. We can rewrite the algorithm to perform the tests in a nondeterministic order by replacing that $^+$CAL code with

```
    j := 1..N ;
l8: while j ≠ {} do                  \* {} is the empty set.
      with p ∈ j do when ~b[p] ;     \* ~ is logical negation.
                    j := j \ {p} ;   \* \ is set difference.
      end with ;
   end while
```

(If you don't know the meaning of the operator "`\`", look it up in the index.) The statement

`with` *id* $\in$ *S* `do` *body*

sets *id* to a nondeterministically chosen element of the set *S* and then executes *body*. (In model-checking mode, TLC will check the algorithm for all possible choices of *id*.) Replacing *id* $\in$ *S* with *id* = *exp* causes the body to be executed with *id* equal to the current value of *exp*.

A multiprocess algorithm can have multiple "`process` / `end process`" sections. The statement

`process` *Name* = *e*

begins a single process named *Name* with identifier *e*. Note that *Name* is an arbitrary name that you give to the process; *e* is an expression. Changing *Name* has no effect on the algorithm, but changing the process's identifier *e* can make a difference. Different processes must have different identifiers. Moreover, the identifiers of all processes should have the same "type"—for example, they should all be integers or all be strings or all be sets of records.

The safety property that algorithm *FastMutex* should satisfy is mutual exclusion, meaning that at most one process can be in its critical section

at any one time. For the +CAL version, this means that no two processes can be at the statement labeled `cs`. An *invariant* is an assertion that is true in every state that can occur during an execution of the algorithm. Mutual exclusion is the invariance of the assertion "no two processes are at statement `cs`". We can tell TLC to check that this assertion is an invariant. But first, we must know how to express the assertion in TLA$^+$.

The TLA$^+$ translation introduces a new variable *pc* whose value is the label of the next statement to be executed. For the uniprocess algorithm *EuclidAlg*, the value of *pc* is the string "lp" iff the next statement to be executed is the `while`. The algorithm has terminated iff $pc = \text{"Done"}$. For a multiprocess algorithm, the variable *pc* is a function whose domain is the set of process identifiers. In algorithm *FastMutex*, a process i is at statement `cs` iff $pc[i]$ equals "cs". Mutual exclusion is therefore asserted by the invariance of the predicate *Mutex*, defined by

$$\begin{aligned} &\mathit{Mutex} \triangleq \\ &\quad \forall\, i, k \in 1 \,..\, N : (i \neq k) \Rightarrow \neg((pc[i] = \text{"cs"}) \land (pc[k] = \text{"cs"})) \end{aligned}$$

(The operators like $\forall$, $\Rightarrow$, $\neg$, and $\land$ are explained in Section 5.3 on page 42. Section 3.8 on page 27 explains why we could not use the identifier j instead of k in the $\forall$ expression.)

TLA$^+$ allows a definition to refer only to variables and operators that have already been defined or declared. Since the definition of *Mutex* uses the variable *pc*, which is declared by the translation of the algorithm, this definition must come after the translation—in other words, after the "`END TRANSLATION`" line.

We tell TLC to check the invariance of *Mutex* by adding the following line to the configuration file.

```
INVARIANT Mutex
```

We then run TLC as before, with the command

```
java tlc.TLC FastMutex
```

if the algorithm appears in a module also named *FastMutex*. We tell TLC to check multiple invariants by listing them in the `INVARIANT` statement—for example

```
INVARIANT Mutex TypeCorrectness Inv3
```

The variable *pc* can be used in the algorithm's expressions We could therefore also check mutual exclusion by putting the following `assert` statement in statement `cs`:

$\forall$ i $\in$ 1..N : (i $\neq$ self) $\Rightarrow$ (pc[i] $\neq$ "cs")

("$\forall$" is typed "\A", and "$\Rightarrow$" is typed "=>".)

Invariance checking is discussed further in Section 4.5. Section 4.6 describes how to check liveness properties, which are the generalization of termination.

3 The Language

This section lists the statements and constructs of +CAL and explains their meanings. In doing so, it also describes the language's grammar. A BNF specification of the grammar appears in Section A on page 54 of the appendix. That grammar and most of the examples in this section show statements and other syntactic units all ending with a semicolon. That final semicolon is not required if it is followed by any of the following tokens.

```
begin  do  else  elsif  end  macro  or  procedure  process
```

Before getting to the language description, we need some definitions. A *statement sequence* is a sequence of statements, each ended by a semicolon. For example, the body of a `while` statement consists of a sequence of statements. If there is an `if` statement in that sequence of statements, then its `then` clause consists of a separate sequence of statements. The statements in the `then` clause are not part of the sequence that forms the `while`'s body. (It is the `if` statement, not the statements that occur inside it, that is a statement of the `while`'s body.)

A *control path* is a path through a piece of +CAL code that represents a syntactically possible execution sequence, if we ignore how the statements are executed. For example, in the code

```
a: if FALSE then     goto w ;
                  b: x := 7 ;
                  c: y := 8 ;
   end if ;
d: x := 0 ;
```

there is a control path that goes from the label `a` to the label `c`—even though no execution can actually follow that path.

A *step* is a control path that starts at a label, ends at a label, and passes through no other labels. In the example above, there are two steps beginning at label `a`—one that ends at `b` and one that ends at `d`. Remember that there is an implicit label `Done` at the end of a uniprocess algorithm and at the end of each process in a multiprocess algorithm. An execution of a +CAL algorithm consists of a sequence of executions of steps. Part of a step can never be executed by itself (except for a `print` or `assert` statement, as described below).

3.1 Expressions

The expressions in +CAL algorithms can be any TLA+ expressions that do not contain a +CAL reserved word or symbol such as `begin` or "`||`". You can write arbitrary TLA+ definitions in the module before the "`BEGIN TRANSLATION`" line and use the defined symbols in the algorithm's expressions. Section 5 explains how to write TLA+ expressions and definitions. Table 1 on page 63 and Table 2 on page 64 provide a convenient summary.

You are probably used to programming languages that allow only simple operators in expressions and allow variables to have only simple values. In +CAL, the following statement assigns to `x` a record whose `a` component is the set of integers from 1 to `N` and whose `bcd` component is the set of all prime numbers less than or equal to `N`.

```
x := [a   ↦ 1..N,
      bcd ↦ {i ∈ 2..N : ∀ j ∈ 2..(i-1) : i % j ≠ 0} ]
```

It may be a while before you learn how to take advantage of +CAL's powerful expression language.

TLA+ has the general rule that an identifier cannot be assigned a new meaning if it already has a meaning. Thus, the identifier i cannot be used as a bound variable in an expression like

$$[i \in 1 \,..\, N \mapsto \text{FALSE}]$$

if it already has a meaning—for example, if i is an algorithm variable. Assigning a new meaning to a symbol can result in a "multiply-defined symbol" syntax error in the algorithm's TLA+ translation.

3.2 The Statements

The examples in Section 2 contain most +CAL statements. Here is a complete list of all the statements that can appear in the body of an algorithm, process, or procedure, along with the rules for labels that pertain to each of them. The labeling rules are also all listed in Section 3.7 below.

3.2.1 Assignment

An assignment is either an assignment to a variable such as

```
y := A + B
```

or else an assignment to a component, such as

```
x.foo[i+1] := y+3
```

If the current value of `x` is a record with a `foo` component that is a function (array), then this assignment sets the component `x.foo[i+1]` to the current value of `y+3`. The value of this assignment is undefined if the value of `x` is not a record with a `foo` component, or if `x.foo` is not a function. Therefore, if such an assignment appears in the code, then `x` will usually be initialized to an element of the correct "type", or to be a member of some set of elements of the correct type. For example, the declaration

```
variable x ∈ [bar : BOOLEAN,
              foo : [1..N → {"on", "off"}] ] ;
```

asserts that initially `x` is a record with a `bar` component that is a Boolean (equal to `TRUE` or `FALSE`) and a `foo` component that is a function with domain `1..N` such that `x.foo[i]` equals either "on" or "off" for each `i` in `1..N`. (The symbol "→" is typed "`->`".)

An *assignment statement* consists of one or more assignments, separated by "`||`" tokens, ending with a semicolon. An assignment statement containing more than one assignment is called a *multiple assignment*. A multiple assignment is executed by first evaluating the right-hand sides of all its assignments, and then performing those assignments from left to right. For example, if `i` = `j` = 3, then executing

```
x[i] := 1 || x[j] := 2
```

sets `x[3]` to 2.

Assignments to the same variable cannot be made in two different assignment statements within the same step. In other words, in any control path, a label must come between two statements that assign to the same variable. However, assignments to components of the same variable may appear in a single multiple assignment, as in

```
x.foo[7] := 13 || y := 27 || x.bar := x.foo ;
```

3.2.2 If

The `if` statement has its usual meaning. The statement

`if` *test* `then` *t_clause* `else` *e_clause* `end if ;`

is executed by evaluating the expression *test* and then executing the *t_clause* or *e_clause* depending on whether *test* equals TRUE or FALSE. The `else`

clause is optional. An `if` statement must have a `then` clause and may have zero or more `elsif...then` clauses optionally followed by an `else` clause. It must be ended by `end if;`. For example, the following two `if` statements are equivalent.

```
if x > 0                        if x > 0
  then x := 0;                    then x := 0
  elsif y > 0 then y := 0;        else if y > 0 then y := 0
  else z := 0;                                  else z := 0;
end if;                                 end if;
                                 end if;
```

An `if` statement that contains a `call`, `return`, or `goto` statement or a label within it must be followed by a labeled statement. (A label on the `if` statement itself is not considered to be within the statement.)

3.2.3 Either

The `either` statement has the form:

```
either clause_1
    or clause_2
     ⋮
    or clause_n
end either ;
```

It is executed by nondeterministically choosing any $clause_i$ that is executable and executing it. The `either` statement can be executed iff at least one of those clauses can be executed. If any $clause_i$ contains a `call`, `return`, or `goto` statement or a label, then the `either` statement must be followed by a labeled statement. The statement

```
if test then t_clause
       else e_clause
end if ;
```

is equivalent to the following, where the `when` statement is explained in Section 3.2.5 on the next page.

```
either when   test ;  t_clause
    or when ¬ test ;  e_clause
end either ;
```

3.2.4 While

The `while` statement has its usual meaning. The statement

`lb :` `while` *test* `do` *body* `end while ;`

is executed like the following `if` statement, where the `goto` statement is explained in Section 3.2.11 on page 22.

`lb :` `if` *test* `then` *body* `; goto lb ; end if ;`

A `while` statement must be labeled. However, the statement following a `while` statement need not be labeled, even if there is a label in its *body*.

3.2.5 When

A step containing the statement `when` *expr* can be executed only when the value of the Boolean expression *expr* is `TRUE`. Although it usually appears at the beginning of a step, a `when` statement can appear anywhere within the step. For example, the following two pieces of code are equivalent.

```
a : x := y + 1 ;          a : when y + 1 > 0 ;
    when x > 0 ;              x := y + 1 ;
b : ...                   b : ...
```

The step from `a` to `b` can be executed only when the current value of `y+1` is positive. (Remember that an entire step must be executed; part of a step cannot be executed by itself.)

3.2.6 With

The statement

`with` *id* $\in$ *S* `do` *body* `end with ;`

is executed by executing the statement sequence *body* with identifier *id* equal to a nondeterministically chosen element of S. (The symbol $\in$ is typed "`\in`".) Execution is impossible if S is empty. This `with` statement is therefore equivalent to

`when` $S \neq$ `{} ;` `with` *id* $\in$ *S* `do` *body* `end with ;`

The two statements

`with` *id* `=` *expr* `do` ... `with` *id* $\in$ `{`*expr*`}` `do` ...

are equivalent. (The expression {*expr*} equals the set containing a single element equal to *expr*.)

In general, a `with` statement has the form

`with` $id_1 \star expr_1$ `;` ... `;` $id_n \star expr_n$ `do` *body* `end with ;`

where each $\star$ may be either $=$ or $\in$. This statement is equivalent to

```
with id_1 ⋆ expr_1 do ... with id_n ⋆ expr_n
   do body end with ... end with ;
```

The body of a `with` statement may not contain a label.

3.2.7 Skip

The statement `skip;` does nothing.

3.2.8 Print

Execution of the statement

`print` *expr* `;`

is equivalent to `skip`, except it causes TLC to print the current value of *expr*. TLC may print the value even if the step containing the `print` statement is not executed because of a `when` statement that appears later in the step.

An algorithm containing a `print` statement must be in a module that EXTENDS the *TLC* module.

3.2.9 Assert

The statement

`assert` *expr* `;`

is equivalent to `skip` if expression *expr* equals TRUE. If *expr* equals false, executing the statement causes TLC to produce an error message saying that the assertion failed and giving the location of the `assert` statement. TLC may report a failed assertion even if the step containing the `assert` statement is not executed because of a `when` statement that appears later in the step.

An algorithm containing an `assert` statement must be in a module that EXTENDS the *TLC* module.

3.2.10 Call and Return

The `call` and `return` statements are described below in Section 3.4 on page 23.

3.2.11 Goto

Executing the statement

`goto` *lab* `;`

ends the execution of the current step and causes control to go to the statement labeled *lab*. In any control path, a `goto` must be immediately followed by a label. (Remember that the control path by definition ignores the meaning of the `goto` and continues to what is syntactically the next statement.)

If is legal for a `goto` to jump into the middle of a `while` or `if` statement, but this sort of trickery should be avoided.

3.3 Processes

A multiprocess algorithm contains one or more processes. A process begins in one of two ways:

`process` *ProcName* $\in$ *IdSet*

`process` *ProcName* `=` *Id*

The first form begins a *process set*, the second an individual process. The identifier *ProcName* is the process or process set's name. The elements of the set *IdSet* and the element *Id* are called *process identifiers*. The process identifiers of different processes in the same algorithm must all be different. This means that the semantics of TLA$^+$ must imply that they are different, which intuitively usually means that they must be of the same "type". (For example, the semantics of TLA$^+$ does not specify whether or not a string may equal a number.) For execution by TLC, this means that all process identifiers must be comparable values, as defined on page 264 of the TLA$^+$ book [2].

The name *ProcName* has no significance; changing it does not change the meaning of the `process` statement in any way. The name appears in the TLA$^+$ translation, and it should be different for different `process` statements

As explained above in Section 2.6 on page 10, the `process` statement is optionally followed by declarations of local variables. The process body is

begun by "`begin`" and ended by "`end process`". Its first statement must be labeled. Within the body of a process set, `self` equals the current process's identifier.

A multiprocess algorithm is executed by repeatedly choosing an arbitrary process and executing one step of that process, if that step's execution is possible. Execution of the process's next step is impossible if the process has terminated, if its next step contains a `when` statement whose expression equals FALSE, or if that step contains a statement of the form "`when` $x \in S$" and S equals the empty set. As explained in Section 2.6 on page 10, fairness conditions may be specified on the choice of which processes' steps are to be executed.

3.4 Procedures

An algorithm may have one or more procedures. If it does, the algorithm must be in a TLA$^+$ module that EXTENDS the *Sequences* module.

The algorithm's procedures follow its global variable declarations and `define` section (if any) and precede the `begin` of a uniprocess algorithm or the first process of a multiprocess algorithm. A procedure named *PName* begins

`procedure` *PName* `(` $param_1$ `, ... ,` $param_n$ `)`

where the identifiers $param_i$ are the *formal parameters* of the procedure. These parameters are treated as variables and may be assigned to. As explained in Section 4.5 on page 35, there may also be initial-value assignments of the parameters. Those initial values are needed by TLC when checking termination or liveness; they do not affect the algorithm's execution.

The `procedure` statement is optionally followed by declarations of variables local to the procedure. These have the same form as the declarations of global variables, except that initializations may only have the form "*variable* `=` *expression*". The procedure's local variables are initialized on each entry to the procedure.

Any variable declarations are followed by the procedure's body, which is begun by "`begin`" and ended by "`end procedure`". The body must begin with a labeled statement. There is an implicit label `Error` immediately after the body. If control ever reaches that point, then execution of either the process (multiprocess algorithm) or the complete algorithm (uniprocess algorithm) halts.

A procedure *PName* can be called by the statement

`call` *PName* `(` $expr_1$ `, ... ,` $expr_n$ `) ;`

Executing this call assigns the current values of the expressions $expr_i$ to the corresponding parameters $param_i$, initializes the procedure's local variables, and puts control at the beginning of the procedure body.

A `return` statement assigns to the parameters and local procedure variables their previous values—that is, the values they had before the procedure was last called—and returns control to the point immediately following the `call` statement.

The `call` and `return` statements are considered to be assignments to the procedure's parameters and local variables. In particular, they are included in the rule that a variable can be assigned a value by at most one assignment statement in a step. For example, if x is a local variable of procedure P, then a step within the body of P that (recursively) calls P cannot also assign a value to x.

For a multiprocess algorithm, the identifier `self` in the body of a procedure equals the process identifier of the process within which the procedure is executing.

The `return` statement has no argument. A $^+$CAL procedure does not explicitly return a value. A value can be returned by having the procedure set a global variable and having the code immediately following the `call` read that variable. For example, in a multiprocess algorithm, procedure P might use a global variable `rVal` to return a value by executing

```
rVal[self] := ... ;
return ;
```

From within a process in a process set, the code that calls P might look like this:

```
call P(17) ;   lab: x := ... rVal[self] ... ;
```

For a call from within a single process, the code would contain the process's identifier instead of `self`.

In any control path, a `return` statement must be immediately followed by a label. A `call` statement must either be followed in the control path by a label or else it must appear immediately before a `return` statement in a statement sequence.

When a `call` P statement is followed immediately by a `return`, the return from procedure P and the return performed by the `return` statement are both executed as part of a single execution step. When these statements are in the (recursive) procedure P, this combining of the two returns is essentially the standard optimization of replacing tail recursion by a loop.

3.5 Macros

A macro is like a procedure, except that a call of a macro is expanded at translation time. You can think of a macro as a procedure that is executed within the step from which it is called.

A macro definition looks much like a procedure declaration—for example:

```
macro P(s, i) begin when s ≥ i ;
                    s := s - i ;
              end macro ;
```

The difference is that the body of the macro may contain no labels, no `while`, `call`, `return`, or `goto` statement, and no macro call. Macro definitions come right after any global variable declarations and `define` section.

A macro call is like a procedure call, except with the `call` omitted—for example:

```
P(sem, y + 17) ;
```

The translation replaces the macro call with the sequence of statements obtained from the body of the macro definition by substituting the arguments of the call for the definition's parameters. Thus, this call of the `P` macro expands to:

```
when sem ≥ (y + 17) ;
sem := sem - (y + 17) ;
```

When translating a macro call, substitution is syntactic in the sense that the meaning of any symbol in the macro definition other than a parameter is the meaning it has in the context of the call. For example, if the body of the macro definition contains a symbol `q` and the macro is called within a "`with q` $\in \ldots$" statement, then the `q` in the macro expansion is the `q` introduced by the `with` statement.

When replacing a macro by its definition, the translation replaces every instance of a macro parameter *id* in an expression within the macro body by the corresponding expression. Every instance includes any uses of *id* as a bound variable, as in the expression

$$[id \in 1 \mathinner{..} N \mapsto \text{FALSE}]$$

The substitution of an expression like $y + 17$ for *id* here will cause a mysterious error when the translation is parsed. When using $^+$CAL, obey the TLA$^+$ convention of never assigning a new meaning to any identifier that already has a meaning.

3.6 Definitions

An algorithm's expressions can use any operators defined in the TLA$^+$ module before the "`BEGIN TRANSLATION`" line. Since the TLA$^+$ declaration of the algorithm's variables follows that line, the definitions of those operators can't mention any algorithm variables. The $^+$CAL `define` statement allows you to write TLA$^+$ definitions of operators that depend on the algorithm's global variables. For example, suppose the algorithm begins:

```
--algorithm Test
  variables x ∈ 1..N ; y ;
  define zy    ≜ y*(x+y)
         zx(a) ≜ x*(y-a)
  end define ;
  ...
```

(The symbol "$\triangleq$" is typed "`==`".) The operators *zy* and *zx* can then be used in expressions anywhere in the remainder of the algorithm. Observe that there is no semicolon or other separator between the two definitions. Section 5.11 on page 52 describes how to write TLA$^+$ definitions.

The variables that may appear within the `define` statement are the ones declared in the `variable` statement that immediately precedes it and that follows the algorithm name, as well as the variable *pc* and, if there is a procedure, the variable *stack*. Local process and procedure variables may not appear in the `define` statement. The `define` statement's definitions need not mention the algorithm's variables. You might prefer to put definitions in the `define` statement even when they don't have to go there. However, remember that the `define` statement cannot mention any symbols defined or declared after the "`END TRANSLATION`" line; and the symbols it defines cannot be used before the "`BEGIN TRANSLATION`" line.

3.7 Labels

Various rules for where labels must or may not appear have been introduced above. The complete set of rules are:

- The first statement in the body of a procedure, of a process, or of a uniprocess algorithm must be labeled.
- A `while` statement must be labeled.
- A statement S in a statement sequence must be labeled if it is preceded in that sequence by any of the following:

  - A `call` statement, if S is not a `return`.
  - A `return` statement.
  - A `goto` statement.
  - An `if` or `either` statement that contains a labeled statement, a `goto`, a `call`, or a `return` anywhere within it.

- A macro body and the `do` clause of a `with` statement cannot contain any labeled statements.

- In any control path, a label must come between an assignment to a variable x and any other statement that assigns a value to x, including a `call` or `return` that sets x if x is a procedure parameter or local procedure variable.

The implicit labels `Done` and `Error` cannot be used as actual labels.

3.8 The Translation's Definitions and Declarations

This section lists all the identifiers declared and defined in the TLA$^+$ translation of a $^+$CAL algorithm. You may need to know what those identifiers are when writing invariants and liveness properties to check the algorithm. Moreover, as explained on page 17 of Section 3.1, TLA$^+$ does not allow the assignment of a new meaning to an identifier that already has a meaning. Redefining an identifier declared or defined by the translation, or using it as a bound variable, will cause a "multiply-defined identifier" error when the TLA$^+$ module is parsed by the SANY parser, which is invoked by TLC.

The translation of a $^+$CAL algorithm declares the following TLA$^+$ variables:

- Each variable declared either globally or locally within a process or a procedure.

- *pc*

- *stack*, if the algorithm contains one or more procedures.

- Each formal parameter of a procedure.

A multiprocess $^+$CAL algorithm defines each of the following. For a uniprocess algorithm, the "(*self*)" argument is omitted.

- For a multiprocess algorithm, the set *ProcSet* of all process identifiers.

- The tuple *vars* of all variables.
- The initial predicate *Init*. It contains a conjunct for each variable. The conjuncts for global variables precede those for local procedure and process variables. The conjuncts for the variables declared in a single `variable` statement appear in the order in which they are declared. (This order is significant, since the initial value of a variable can depend on the initial values assigned by previous conjuncts.)
- The next-state action *Next* and the complete specification *Spec*.
- For each statement label *Lab*, an action *Lab*(*self*) if the statement is in a procedure or in a process set; otherwise, an action *Lab*. This action is the TLA$^+$ representation of the atomic operation beginning at that label. (Actions and atomic operations are discussed in Section 5.10.1 on page 49.) If the definition is of *Lab*(*self*), then this is the action describing the operation performed by a process *self*, for *self* in *ProcSet*.
- For each procedure P, an action $P(self)$. It is the disjunction of all actions in the procedure executed by a process with identifier *self* in *ProcSet*.
- For each process set named P, an action $P(self)$. It is the disjunction of all actions not in a procedure that are executed by a process with identifier *self* in the process set.
- For each single process named P, an action P that is the disjunction of all actions not in a procedure that are executed by the process.

Because TLA$^+$ does not allow an identifier to be declared or defined multiple times, the translation may rename some of these identifiers to produce a legal TLA$^+$ specification. For example, if the $^+$CAL code declares a variable `x` and also uses `x` as a label, or if it declares `x` as a local variable in two different procedures, then one of the two `x`'s must be renamed. If the translator renames identifiers, then it issues a warning and indicates, in comments placed right after the "`BEGIN TRANSLATION`" line, what renamings have been done.

Identifiers defined or declared in the translation may not be given new meanings in any TLA$^+$ definition that follows the "`END TRANSLATION`" line. For example, if the $^+$CAL algorithm declares a variable `j`, then a definition that follows the translated algorithm cannot contain the expression

$$\forall i, j \in 1..N : (i \neq j) \Rightarrow \neg((pc[i] = \text{“cs”}) \land (pc[j] = \text{“cs”}))$$

that redeclares the identifier j. Such a re-use of an identifier causes a "multiply-defined identifier" error when the TLA$^+$ module is parsed.

4 Checking the Algorithm

Sections 2.3–2.5 above tell you how to use the translator and TLC model checker to check an algorithm. This section explains more about the translator and TLC. Only the commonly used features of TLC are described. You'll have to consult Chapter 14 of the TLA$^+$ book for a more complete description of what TLC can do. Also, check the document *Current Versions of the TLA$^+$ Tools* on the TLA$^+$ tools web page for recently-added features. That page can be found from the main TLA$^+$ web page, a link to which is at `http://lamport.org`.

TLC takes as input a TLC module and a configuration file. A module named M is in file M`.tla`, and its configuration file is named M`.cfg`. Running the $^+$CAL translator on file M rewrites the file M`.cfg`, creating it anew if that file doesn't already exist. (You can keep the translator from writing or rewriting the configuration file with the `-nocfg` option.) Normally, you will let the translator create the configuration file and then add anything else needed to check the algorithm. If you put those additions where the file tells you to, they will be preserved when the translator rewrites the file.

4.1 Running the Translator

Running the translator is simple; Section 2.3 on page 7 explains how to do it. Section 2.5 on page 9 describes the translator's `-termination` option. The other options you are likely to use are ones that specify fairness properties; they are described in Section 4.6 on page 35. To find out about all the available options, run the translator with the `-help` option by typing

```
java pcal.trans -help
```

The one part of using the translator that can be tricky is understanding its messages. The only warning message whose meaning may not be obvious is

```
Warning: symbols were renamed.
```

It means that the translator has renamed one or more symbols used in the algorithm. Section 3.8 on page 27 explains why this was done. The renamings are listed in the comments within the translation, right after the "`BEGIN TRANSLATION`" line.

There are two kinds of translator error messages that can be mysterious. The first is one saying that the translator was expecting to find a certain token and didn't. For example, the missing semicolon at the end of the first line of

```
L1:  a := b + c
L2:  f[x] := c
```

produces the error message

```
-- Expected ";" but found ":="
   line ..., column ....
```

where the line and column numbers indicate the location of the second "`:=`". We might expect the translator to complain when it finds "`b + c`" followed by "`L2`", since no legal expression can begin $b + c\ L2$. However, the translator does not try to parse expressions. It leaves that task to the SANY parser, which is used by TLC. Instead, upon seeing the "`:=`" in the first statement, the translator just assumes that everything until the next reserved symbol is part of the assignment statement's expression. It discovers that something is wrong when it finds the expression ended by "`:=`".

The lesson to be learned from this example is that the source of an error can come well before the location where the error is reported. If you can't find the cause of an error, try narrowing in on it by running the translator with sections of the code commented out. (You can do this by bracketing the code with `(*` and `*)`, even if it contains comments.)

The second class of error that can be mysterious is one caused by omitting a needed label. There are two error messages indicating such an error:

```
-- Statement at ... must have a label
-- Multiple assignment to ...
```

Section 3.7 on page 26 gives the rules for where labels are needed. The second message indicates a violation of the rule that, on any control path, two separate statements that assign a value to the same variable must be separated by a label. If you are mystified by this message, it may be because you've forgotten that `call` and `return` statements assign values to a procedure's parameters and local variables.

4.2 Specifying the Constants

Most algorithms are written in terms of constant parameters, declared in the TLA$^+$ module with a CONSTANT statement. Those constants must be given specific values with a `CONSTANT` statement in the configuration file. You can also assign new meanings to defined constants and constant operators for the purpose of model checking. For example, an algorithm might contain a statement

```
with i ∈ Nat do ...
```

where *Nat* is defined by the standard *Naturals* module to be the set of all natural numbers. TLC cannot check an algorithm that requires it to enumerate an infinite set like *Nat*. However, you could use the `CONSTANT` statement in the configuration file to tell TLC to substitute a finite set of numbers for *Nat*.

A `CONSTANT` statement in the configuration file consists of the keyword `CONSTANT` followed by a sequence of *assignments* and/or *replacements*, such as

```
CONSTANT  N = 13   Proc = {p1, p2, p3}   gcd <- fastGcd
```

This statement directs TLC to perform three substitutions:

- The assignment `N = 13` tells TLC to substitute the number 13 for N, where N is a constant either declared or defined in the module.
- The assignment to `Proc` tells TLC to substitute for *Proc* the set consisting of the three *model values* $p1$, $p2$, and $p3$. A model value m is a special type of value that TLC assumes is unequal to any value it encounters other than m itself.
- The replacement `gcd <- fastGcd` tells TLC to substitute for *gcd* the value or operator *fastGcd*, which must be defined in the module. For example, *gcd* might be the operator defined as on page 9, and *fastGcd* might be an alternative definition that TLC can compute more efficiently. You could use *gcd* in the algorithm because its definition is easy to understand, but speed up the checking by having TLC compute *fastGcd* instead.

An assignment in a `CONSTANT` statement has the form $Id = exp$, where Id is an identifier and exp is a *simple expression*. A simple expression is a number, a string, a model value, or a finite set $\{e_1, \ldots, e_n\}$ where each e_i is a simple expression. A replacement has the form $Id1$ `<-` $Id2$, where $Id1$ and $Id2$ are identifiers, and $Id2$ is defined in the module.

TLA$^+$ allows you to declare a constant parameter to be an operator that takes one or more arguments. For example, the declaration

CONSTANT $Foo(_)$

declares *Foo* to be an unspecified operator that takes a single argument. The configuration file must use a replacement ("`<-`") in a `CONSTANT` statement to substitute an operator defined in the module for the parameter *Foo*.

4.3 Constraints

TLC tries to generate all reachable states of the algorithm. It does this by repeatedly finding all states that can be reached with a single step from a reachable state that it has already found, starting with all possible initial states. It will run forever if there are an infinite number of reachable states.

Some algorithms have infinitely many reachable states because they have counters or queues that can grow without bound. You can limit the reachable states that TLC examines by using a *constraint*, which is an arbitrary Boolean expression. If TLC finds a reachable state s that does not satisfy the constraint, then it will not look for states that can be reached from s. For example, putting in the TLA$^+$ module the definition

$$Xsmall \triangleq x < 17$$

and putting in the configuration file the statement

```
CONSTRAINT Xsmall
```

causes TLC to find only those reachable states that are either initial states or are reachable by a sequence of states all having x less than 17.

4.4 Understanding TLC's Output

When TLC is run, the first thing it does is call the SANY program to parse the TLA$^+$ module. Parsing may reveal a syntactic error in the module. The error can be either in the part of the module that you wrote or in the part written by the translator. The translator does not parse expressions, leaving it to SANY to find most errors in the algorithm's expressions. You should be able to figure out the problem because the translation copies your expressions pretty much the way you typed them, except for the following changes.

- Some variables are primed.
- Variables local to a process are turned into functions (arrays) that take an additional argument. For example, in algorithm *FastMutex* of Figure 2 on page 12, each occurrence of the local variable j is replaced by $j[self]$.
- An assignment to an element of a function or record variable is rewritten as an assignment to the variable using the TLA$^+$ EXCEPT construct explained in Section 5.7 on page 46.

- Variables may be renamed, as explained in Section 3.8 on page 27.

If the parser complains that an identifier has been multiply defined, it may mean that you have redefined or used as a bound variable an identifier that is defined or declared in the algorithm's translation. This problem is discussed above in Section 3.8 on page 27.

Occasionally, it may be difficult to figure out the cause of a parsing error. In that case, try inserting a "`==`$\cdots$`==`" line to prematurely end the module in different places until you find the definition or statement that is causing the error. You can run the parser without running TLC by typing

`java tlasany.SANY` *file*

If TLC successfully parses the module and finds no problem with the configuration file, then it begins executing the algorithm. There are two kinds of errors it can find: (i) an `assert` statement is executed when its expression is false or some property that you asked TLC to check is not satisfied, or (ii) the algorithm is trying to evaluate a meaningless expression such as *foo.bar* if *foo* does not equal a record. In the first case, TLC tells you which assertion or property is violated. In the second, it usually prints out the stack of nested expressions it was executing when it found the error; but in some cases it just prints the unhelpful message "`null`".

For any error, TLC prints out the sequence of states reached in the execution up to the point at which the error occurred. A state consists of an assignment of values to all the variables. TLC prints most values as ordinary TLA$^+$ expressions, as described in Section 5. However, functions are described in terms of the operators @@ and $:>$ that are defined in the *TLC* module. The expression

$$d_1 :> e_1 \ @@ \ d_2 :> e_2 \ @@ \ \ldots \ @@ \ d_n :> e_n$$

equals the function f with domain $\{d_1, \ldots, d_n\}$ such that $f[d_i] = e_i$ for each i in $1 \,..\, n$.

It can sometimes be quite difficult to figure out the cause of an error from TLC's error message. In that case, you can debug by inserting `print` statements in the algorithm. You can also use the *Print* operator in the algorithm's expressions or in the invariants that TLC is checking. The operator *Print* is defined in the *TLC* module so $Print(pval, val)$ equals val, but TLC prints the value of *pval* when evaluating it.

4.5 Invariance Checking

The examples in Section 2 explain how to use TLC to check invariance of a formula—meaning that the formula is true in all states reached in any execution of the algorithm. An important example of invariance is type correctness. In ordinary typed programming languages, type correctness is a syntactic condition. Because +CAL is typeless, type correctness is a property of the algorithm, asserting that the value of each variable is an element of the proper set. For example, we say that a variable `p` has type *prime number* iff the value of `p` is always a prime number—in other words, iff the following formula is an invariant, where `Nat` is the set of natural numbers.

```
p ∈ {i ∈ Nat : ∀ j ∈ 2..(i-1) : i % j ≠ 0}
```

(If you don't understand this invariant now, you should after reading Section 5.) TLC can check if this formula is an invariant. Like type checking in ordinary programs, checking type correctness is a good way to find simple errors in a +CAL algorithm.

For an algorithm to be type correct, the initial values of its variables must be of the right "type". If no initial value is specified for a variable, its default initial value is `{}` (the empty set). If `{}` is not a type-correct value for the variable, then the algorithm will not be type correct unless the variable is properly initialized. Among the variables whose type you might want to check are the procedure parameters. An algorithm can assign initial values to a procedure's formal parameters as indicated in this example:

```
procedure  Foo (p1 = 0, p2 = {"a", "b"})
```

Like a procedure variable's declaration, the initial-value declaration of a formal parameter *p* must be of the form *p* `=` *expression*.

Since a procedure's formal parameters are set equal to the corresponding arguments when the procedure is called, their initial values do not affect the execution. Those initial values serve only to ensure that the corresponding variables in the TLA$^+$ specification always have values of the correct type.

4.6 Termination and Liveness

We saw in Section 2.5 how to check termination of a uniprocess algorithm. Termination is a special case of a general class of properties called *liveness* properties, which assert that something must eventually happen. We can use TLC to check more general liveness properties of an algorithm. As with

termination, checking liveness requires that each TLA$^+$ variable be initialized to a value of the proper type. See the discussion of type correctness in Section 4.5 on the preceding page.

An algorithm satisfies a liveness property only under some assumption—usually a fairness assumption. There are many possible choices of fairness conditions that we may want to assume. They can be expressed with the TLA$^+$ weak and strong fairness operators, WF and SF. A common fairness assumption for multiprocess algorithms is weak fairness of each process's execution. This means that, for each process P, if control in P is at an operation that it is always possible to execute, then P must eventually execute that operation. Fairness is discussed in Section 5.10.1 on page 49.

The TLA$^+$ temporal operators used to express fairness and liveness are described in Section 5.10 on page 49. Temporal properties are subtle and can be hard to understand. Chapter 8 of the TLA$^+$ book discusses these properties in more detail.

As an example, here is how you can check that algorithm *FastMutex* of Section 2.6 satisfies the following property.

> Assuming weak fairness of each process's execution, infinitely often there is a process in its critical section.

(The algorithm is deadlock and livelock free but not starvation free; it is possible for all but one process never to enter its critical section.) We run the translator with the command

```
java pcal.trans -wf FastMutex
```

The `-wf` option instructs the translator to define the specification *Spec* so it asserts weak fairness for each process. In the TLA$^+$ module, we define the formula

$$Liveness \triangleq \Box\Diamond(\exists\, i \in 1 \,..\, N : pc[i] = \text{“cs”})$$

which asserts that some process is infinitely often in its critical section. (The temporal operators $\Box$ and $\Diamond$ are explained in Section 4.6 on 35.) We tell TLC to check this property by putting the following statement in the configuration file.

```
PROPERTY Liveness
```

We then run TLC in model-checking mode.

If the liveness property you want to check is termination, you can just use the translator's `-termination` option. It instructs the translator to put the

appropriate `PROPERTY` statement into the configuration file. If no fairness option is given, the `-termination` option assumes weak fairness, as if the `-wf` option were specified. Use the `-nof` option to specify that no fairness condition is to be assumed.

There are many different fairness assumptions one might want to assert about an algorithm. The translator provides the following additional options for asserting fairness assumptions.

`-sf` Strong fairness of each process's execution.

`-wfNext` Weak fairness of the entire algorithm's next-state action. (This is a weaker assumption than weak fairness for each process.)

`-nof` Assumes no fairness assumption. This is the default, unless the `-termination` option is chosen.

For uniprocess algorithms, the fairness conditions asserted by the `-wf`, `-sf`, and `-wfNext` options are all equivalent. Algorithm *FastMutex* satisfies property *Liveness* under the assumption of weak fairness of the entire algorithm's next-state action.

You may want to use a different liveness assumption than the ones provided by these translator options. For example, in a mutual exclusion algorithm like *FastMutex*, we don't really want to assume weak fairness of a process's execution. That assumption forbids a process from halting in its noncritical section, but the mutual exclusion problem allows it to do so. It is not clear whether a process should be allowed to halt in its critical section, but let's suppose that it is. What we then want to prove is that, assuming weak fairness of each process's execution outside its noncritical and critical section, if infinitely often there is a process trying to enter its critical section, then infinitely often there is a process in its critical section. From the list in Section 3.8 of actions defined by the translation, or by examining the actual translation described in Section B.1 of the appendix, and from the meaning of the WF operator described in Section 5.10, we find that this fairness assumption is expressed by the formula

$$\begin{aligned} \mathit{Fairness} \;\triangleq\; \forall\, i \in 1\,..\,N : \;& \wedge\, \mathrm{WF}_{\mathit{vars}}(\mathit{start}(i)) \\ & \wedge\, \mathrm{WF}_{\mathit{vars}}(l1(i)) \\ & \wedge\, \mathrm{WF}_{\mathit{vars}}(l2(i)) \\ & \quad \ldots \\ & \wedge\, \mathrm{WF}_{\mathit{vars}}(l12(i)) \end{aligned}$$

The specification with fairness is therefore

$$\mathit{FairSpec} \;\triangleq\; \mathit{Spec} \;\wedge\; \mathit{Fairness}$$

To get TLC to use this as the specification, we put the following statement in the configuration file.

```
SPECIFICATION FairSpec
```

If you run the translator after adding this statement, the translator will remove any `SPECIFICATION` statement it may have added. Otherwise, you will have to remove it yourself.

Since a process i is trying to enter its critical section iff $pc[i] \neq$ "ncs", the liveness property we want to check is

$$CondLiveness \triangleq (\Box\Diamond\exists\, i \in 1 \,..\, N : pc[i] \neq \text{"ncs"}) \Rightarrow Liveness$$

where *Liveness* was defined above. We get TLC to check this property by putting the following statement in the configuration file.

```
PROPERTY CondLiveness
```

Remember that when checking any liveness property, the algorithm must initialize all variables with values of the proper type. Liveness checking (including termination) is slower than invariance checking, and TLC cannot check liveness on as large a model as it can check invariance.

4.7 Additional TLC Features

4.7.1 Deadlock Checking

An algorithm is *deadlocked* if it has not terminated, but it can take no further step. A process has terminated if it has reached the end of its code, so control is at the implicit `Done` label that ends its body.

The most likely way for a uniprocess algorithm to deadlock is for a procedure call to "fall off the end" without executing a `return` statement—that is, for it to reach the implicit label `Error` that ends the procedure body.

A multiprocess algorithm is deadlocked if no process can take a step, but some process has not terminated. The usual way for this to happen is for each processes to be waiting either at a `when` statement whose expression is false or at a statement of the form "`with` $x \in S \ldots$" when S equals the empty set.

Deadlock is normally an error and is reported by TLC. However, sometimes an algorithm is supposed to halt in a state in which not all processes have reached the end of their code. To stop TLC from checking for deadlock, run it with the `-deadlock` option.

4.7.2 Multithreading

TLC can execute with multiple threads to take advantage of a multiprocessor computer. Running it with the option `-workers` w tells it to use w threads. The number w should be at most equal to the number of actual processors the computer has. (Running it with more threads than there are processors can slow TLC down.) In theory, using w processors can speed up TLC's computation of the set of reachable states by a factor of almost w. In practice, the speedup depends on the quality of the Java runtime's implementation of multithreading. TLC's algorithm for checking liveness is single-threaded, so additional worker threads will not speed up that part of TLC's execution.

4.7.3 Symmetry

Many algorithms are symmetric in a set of values. For example, the fast mutual exclusion algorithm described in Section 2.6 on page 10 is symmetric in the set of process identifiers. This means that, given any possible execution of the algorithm, permuting the set of identifiers of the processes yields a possible execution. Exactly what symmetry means is explained in Section 14.3.4 on page 245 of the TLA$^+$ book.

We can get TLC to take advantage of symmetry in a set *SSet* to speed up its checking of an algorithm. It does this by ignoring any new state it finds that is the same as a state it has already found under a permutation of the elements of *SSet*. We tell it to assume symmetry in *SSet* as follows. We first declare *SSet* to be a parameter of the algorithm, declared in a CONSTANT statement. In the TLA$^+$ module, we define

$$Perms \triangleq Permutations(SSet)$$

where *Permutations* is defined in the *TLC* module, which must be imported in an EXTENDS statement. In the configuration file's `CONSTANT` statement, we substitute a set of model values for *SSet*—for example, with

```
CONSTANT SSet = {p1, p2, p3}
```

(Model values are explained on page 32.) We also add the following statement to the configuration file.

```
SYMMETRY Perms
```

TLC can use symmetry only for a set of model values. For TLC to take advantage of the symmetry of algorithm *FastMutex*, that algorithm would

have to be rewritten to use an arbitrary set of process identifiers instead of the set of numbers $1 \,..\, N$.

We can also tell TLC to assume symmetry in more than one set of values. For example, an algorithm might be symmetric in both a set *SSet* of processes and a set *Val* of values. We proceed as before, declaring *SSet* and *Val* to be parameters, having TLC substitute sets of model values for them, and using the same `SYMMETRY` statement. However, we define *Perm* by

$$Perms \triangleq Permutations(SSet) \cup Permutations(Val)$$

When we instruct TLC to assume that an algorithm is symmetric, it does not check whether the algorithm really is symmetric. That's our responsibility.

Do not tell TLC both to assume symmetry and to check liveness. The interaction of a symmetry assumption with TLC's algorithm for checking liveness is subtle. It's hard to determine if liveness checking will produce correct results when symmetry is assumed.

5 TLA$^+$ Expressions and Definitions

We now describe the TLA$^+$ operators with which $^+$CAL expressions are built. They are also listed with brief explanations in Tables 1–3 on pages 63–64. Only TLA$^+$ operators that can be evaluated by TLC are given.

We show the typeset versions of all expressions. Table 5 on page 65 shows how symbols with no obvious ASCII representation are typed. TLA$^+$ keywords are typed with upper-case letters, so TRUE is typed as `TRUE`.

5.1 Numbers

Non-negative integers are typed in the usual way as strings of decimal digits. The standard module *Naturals* defines the following standard operators on integers:

$$+ \quad - \quad * \quad \hat{} \text{ (exponentiation)} \quad < \quad > \quad \leq \quad \geq \quad \% \quad \div \quad ..$$

where $-$ is subtraction, not the unary negation operator. The expression a^b is typed `a^b`. The operator " $..$ " is defined so $a \,..\, b$ is the set of all integers c such that $a \leq c \leq b$. The modulus operator $\%$ and the integer division operator $\div$ are defined so that, for any integer a and positive integer b, the value of $a \,\%\, b$ is in $0 \,..\, (b-1)$ and

$$a \;=\; b * (a \div b) \;+\; (a \,\%\, b)$$

The *Naturals* module also defines *Nat* to be the set of all natural numbers (non-negative integers).

The *Integers* module defines everything the *Naturals* module does plus the unary "$-$" operator and the set *Int* of all integers. You are unlikely to use *Nat* or *Int* in an algorithm, but you might very well write something like $n \in Int$ in a type-correctness invariant (Section 4.5).

5.2 Strings

Strings are enclosed in double-quotes (`"`), so the string "`abc`" is typed `"abc"`. The following pairs of characters are used to represent certain special characters in strings.

`\"`	`"`	`\t`	tab	`\f`	form feed
`\\`	`\`	`\n`	line feed	`\r`	carriage return

A string is defined in TLA$^+$ to be a sequence of characters, but TLC does not treat them as first-class sequences. TLC treats strings as a primitive data

F	G	$F \land G$
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

F	G	$F \lor G$
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

F	G	$F \Rightarrow G$
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	TRUE
FALSE	FALSE	TRUE

F	G	$F \equiv G$
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	TRUE

Figure 3: Truth tables for the binary Boolean operators.

type, except that the operators "$\circ$" (concatenation) and *Len* (length) defined in the standard *Sequences* module work properly on them—for example, TLC knows that "ab" $\circ$ "c" equals "abc" and *Len*("abc") equals 3. (These operators are described in Section 5.8 on page 47 below.) Even though sequences in TLA$^+$ are functions, TLC does not regard them as such and it cannot evaluate "abc"[2].

5.3 Boolean Operators

The Boolean values are written TRUE and FALSE. The set BOOLEAN contains these two values. The five propositional operators on Booleans are

$\land$	conjunction (and, typed "/\")	$\Rightarrow$	implication (typed "=>")
$\lor$	disjunction (or, typed "\/)"	$\equiv$	equivalence (typed "<=>" or "\equiv")
$\neg$	negation (not, typed "~")		

The four binary operators are defined by the truth tables of Figure 3 on this page. The operator $\neg$ is defined by

$\neg$TRUE $=$ FALSE $\qquad$ $\neg$FALSE $=$ TRUE

In addition to the usual binary operators $\land$ and $\lor$, TLA$^+$ also allows a bulleted-list notation for conjunctions and disjunctions. For example, the expression

$$\begin{array}{l} \land\ A \\ \land\ \lor\ B \\ \quad \lor\ C \\ \land\ D \end{array}$$

equals $A \land (B \lor C) \land D$. Indentation is used to eliminate parentheses, which can make a complicated formula easier to read. The $\land$ or $\lor$ symbols in a bulleted-list conjunction or disjunction must line up exactly.

Universal and existential quantification over sets of values have the following forms:

$\forall\, x \in S : P(x)$

The expression that equals TRUE if $P(x)$ equals TRUE for all elements x in the set S, and equals FALSE if $P(x)$ equals FALSE for some x in S. Thus, $\forall\, n \in 1 \,..\, 3 : f[n] > y$ is equivalent to

$$(f[1] > y) \land (f[2] > y) \land (f[3] > y)$$

$\exists\, x \in S : P(x)$

The expression that equals TRUE if $P(x)$ equals TRUE for some x in S, and equals FALSE if $P(x)$ equals FALSE for all x in S. Thus, $\exists\, n \in 1 \,..\, 3 : f[n] > y$ is equivalent to

$$(f[1] > y) \lor (f[2] > y) \lor (f[3] > y)$$

In these expressions, the bound identifier x may not already be defined or declared, and it may not occur in the expression S. In the case of S equal to the empty set $\{\}$, these definitions become

$$\forall\, x \in \{\} : P(x) \equiv \text{TRUE}$$
$$\exists\, x \in \{\} : P(x) \equiv \text{FALSE}$$

for any P. TLA$^+$ allows some obvious abbreviations for nested quantifiers. For example,

$$\forall\, x \in S,\, y \in T : F \quad \text{means} \quad \forall\, x \in S : (\forall y \in T : F)$$
$$\exists\, x, y \in S : F \quad \text{means} \quad \exists\, x \in S : (\exists\, y \in S : F)$$

5.4 Sets

Enumerated finite sets are written in the usual way, $\{e_1, \ldots, e_n\}$ being the set containing the elements e_1, ..., e_n. For example, $\{1 + 1, 2 + 2, 4\}$ is the set containing the two elements 2 and 4. (Remember that an element either is or is not an element of a set; it makes no sense to talk about a set containing multiple copies of an element.) As a special case of this notation,

{} is the empty set (the set containing no elements). TLA$^+$ provides the following operators on sets.

$\in$ (membership)	$\cup$ (union)	UNION (big $\bigcup$)
$\subseteq$ (subset)	$\cap$ (intersection)	SUBSET (power set)
$\setminus$ (set difference)		

Here are their definitions:

$e \in S$ Equals TRUE if e is an element of the set S and equals FALSE otherwise.

$S \cap T$ The set of elements in both S and T.

$S \cup T$ The set of elements in S or T (or both).

$S \subseteq T$ True iff every element of S is an element of T.

$S \setminus T$ The set of elements in S that are not in T.

UNION S The union of the elements of S. In other words, a value e is an element of UNION S iff it is an element of an element of S. Mathematicians usually write this as $\bigcup S$.

SUBSET S The set of all subsets of S. Mathematicians sometimes call this the *power set* of S and write it as $\mathcal{P}(S)$ or 2^S.

Mathematicians often describe a set as "the set of all ... such that ...". The following two constructs formalize such a description.

$\{x \in S : P(x)\}$ The subset of S consisting of all elements x satisfying property $P(x)$. For example, the set of all odd natural numbers can be written $\{n \in Nat : n \,\%\, 2 = 1\}$.

$\{e(x) : x \in S\}$ The set of elements of the form $e(x)$, for all x in the set S. For example, $\{2*n + 1 : n \in 1 \,..\, 100\}$ is the set $\{3, 5, 7, \ldots, 201\}$.

In these expressions, the bound identifier x may not already be defined or declared, and it may not occur in the expression S. The construct $\{e(x) : x \in S\}$ has the same generalizations as $\exists x \in S : F$. For example, $\{e(x, y) : x \in S,\ y \in T\}$ is the set of all elements of the form $e(x, y)$, for x in S and y in T.

The standard module *FiniteSets* defines *Cardinality*(S) to be the cardinality (number of elements in) the finite set S.

The expression CHOOSE $x \in S : P(x)$ is defined to equal some arbitrarily chosen value x in the set S such that $P(x)$ equals TRUE. If no such x exists, then the value of that expression is unspecified, and TLC will report an error when evaluating it. The CHOOSE operator is known to logicians as *Hilbert's* ε. This operator cannot be used to introduce nondeterminism in an algorithm. The $^+$CAL statement

```
n := CHOOSE i ∈ 1..7 :  TRUE
```

will assign to `n` the same value every time it is executed. That value is some single unspecified integer in the set $1 \,..\, 7$.

5.5 Functions

What programmers call an array, mathematicians call a function. Intuitively, a function f maps each element d in its domain to the value $f[d]$. If f is not a function or d is not in the domain of f, then the meaning of $f[d]$ is not specified and TLC will report an error if it tries to evaluate that expression.

A function is completely specified by its domain and the value of $f[d]$ for every d in its domain. If f is a function, then DOMAIN f is its domain. The expression $[x \in S \mapsto e(x)]$ equals the function f whose domain is S such that $f[d] = e(d)$ for every d in S. For example,

$$[i \in \{1, 2, 3\} \mapsto 2 * i]$$

is the function *twice* with domain $\{1, 2, 3\}$ such that

$$\mathit{twice}[1] = 1 \qquad \mathit{twice}[2] = 4 \qquad \mathit{twice}[3] = 6$$

Using the operators @@ and $:>$ defined in the *TLC* module, this function can also be written

$$(1 :> 1) \mathbin{@@} (2 :> 4) \mathbin{@@} (3 :> 6)$$

For any sets S and T, the expression $[S \to T]$ is the set of all functions f with domain S such that $f[d]$ is in T for all d in S.

Functions are first-class values, so $f[d]$ can be a function. For example, the function

$$[i \in \mathit{Nat} \mapsto [j \in 1 \,..\, N \mapsto (2 * i) \,\%\, j]]$$

is a function f such that $f[3][x]$ equals $6 \,\%\, x$.

TLA$^+$ also allows functions of multiple arguments. For example,

$$[i \in Nat,\, j \in 1 \,..\, N \;\mapsto\; (2 * i) \,\%\, j]$$

is a function g of two arguments such that $g[3, x]$ equals $6 \,\%\, x$. A function with multiple arguments is actually a function with a single argument that is a tuple. For example, $g[3, x]$ is shorthand for $g[\langle 3, x \rangle]$. (Section 5.8 on the next page discusses tuples.)

5.6 Records

TLA$^+$ provides records that are much like the records (also called structs) of ordinary programming languages. If *exp* is a record-valued expression, then *exp.bar* is the *bar* field of that record. The expression

$$[foo \mapsto 17,\; bar \mapsto \{1, 2, 3\}]$$

equals the record r containing a *foo* field whose value is 17 and a *bar* field whose value is the set $\{1, 2, 3\}$. This record r is an element of the set

$$[foo \;:\; Nat,\; bar \;:\; \text{SUBSET } 1 \,..\, 13]$$

consisting of all records with a *foo* field that is an element of the set *Nat* of natural numbers and a *bar* field that is an element of the set SUBSET $1 \,..\, 13$ of all subsets of the set $1 \,..\, 13$.

In TLA$^+$, a record with fields *foo* and *bar* is actually a function whose domain is the set {"foo", "bar"}. The expression *exp.bar* is shorthand for *exp*["bar"].

5.7 The Except Construct

TLA$^+$ provides an EXCEPT construct for describing a function or record that is almost the same as a given function or record. You will probably not need to use the EXCEPT construct yourself. However, it is used extensively in the TLA$^+$ translation of $^+$CAL programs, so you must know how to interpret it if you want to understand the translation.

If f is a function, then $[f \text{ EXCEPT } ![c] = exp]$ equals the value of f after executing the $^+$CAL statement $f[c] := exp$. Thus, $[f \text{ EXCEPT } ![c] = exp]$ is the function g that is the same as f except that $g[c] = exp$. This function can also be written

$$[x \in \text{DOMAIN } f \;\mapsto\; \text{IF } \; x = c \; \text{ THEN } \; exp \; \text{ ELSE } \; f[x]]$$

Similarly, if r is a record, then $[r \text{ EXCEPT } !.c = exp]$ is the record that equals the value of r after executing $r.c := exp$. In other words, it is the record that is the same as r except that its c field equals exp. Since a record is a function whose domain is a set of strings, $[r \text{ EXCEPT } !.c = exp]$ is the same as $[r \text{ EXCEPT } ![\text{“c”}] = exp]$.

A “!” clause of an EXCEPT construct can be more complicated. For example $[f \text{ EXCEPT } ![c].d[e] = exp]$ is the value of f after executing the statement $f[c].d[e] := exp$. You can check that this equals

$$[f \text{ EXCEPT } ![c] = [f[c] \text{ EXCEPT } !.d = [f[c].d \text{ EXCEPT } ![e] = exp]]]$$

An EXCEPT expression can have multiple “!” clauses. For example, the expression $[f \text{ EXCEPT } ![c] = exp1, ![d].e = exp2]$ equals the value of f after executing the $^+$CAL multiple assignment statement

$$f[c] := exp1 \;||\; f[d].e := exp2$$

Remember that a multiple assignment is executed by first evaluating all the right-hand expressions, then performing the assignments from left to right. This implies that $[f \text{ EXCEPT } ![c] = exp1, ![d].e = exp2]$ is equal to

$$[\,[f \text{ EXCEPT } ![c] = exp1] \text{ EXCEPT } ![d].e = exp2\,].$$

5.8 Tuples and Sequences

A finite sequence is what programmers usually call a *list*. In TLA$^+$, a sequence of length n is the same as an n-tuple, which is defined to be a function with domain $1 \,..\, n$. Finite sequences are written in angle brackets $\langle\ \rangle$. The sequence $\langle e_1, \ldots, e_n \rangle$ is the function s with domain $1 \,..\, n$ such that $s[i]$ equals e_i, for each i in $1 \,..\, n$. Thus, $\langle \text{“a”}, \text{“bc”}, \text{“de”} \rangle[3]$ equals “de”.

Sets of tuples can be described with the Cartesian product operator $\times$. For example, $Nat \times Int \times \{\text{“a”}, \text{“b”}, \text{“c”}\}$ is the set of all triples $\langle x, y, z \rangle$ such that $x \in Nat$, $y \in Int$, and $z \in \{\text{“a”}, \text{“b”}, \text{“c”}\}$.

The standard module *Sequences* defines the following operators:

Seq(S) The set of all sequences of elements of the set S. For example, $\langle 3, 7 \rangle$ is an element of $Seq(Nat)$.

Head(s) The first element of sequence s. For example, $Head(\langle 3, 7 \rangle)$ equals 3.

Tail(s) The tail of sequence s, which consists of s with its head removed. For example, $Tail(\langle 3, 7, \text{“a”} \rangle)$ equals $\langle 7, \text{“a”} \rangle$.

Append(s, e) The sequence obtained by appending element e to the tail of sequence s. For example, *Append*$(\langle 3, 7\rangle, 3)$ equals $\langle 3, 7, 3\rangle$.

$s \circ t$ The sequence obtained by concatenating the sequences s and t. For example, $\langle 3, 7\rangle \circ \langle 3\rangle$ equals $\langle 3, 7, 3\rangle$.

Len(s) The length of sequence s. For example, *Len*$(\langle 3, 7\rangle)$ equals 2.

5.9 Miscellaneous Constructs

An IF expression has the form

IF *bool* THEN *t_expr* ELSE *e_expr*

If *bool* equals TRUE, then this expression equals *t_expr*; if *bool* equals FALSE, then it equals *e_expr*.

The CASE expression

CASE $p_1 \rightarrow e_1 \;\Box\; \ldots \;\Box\; p_n \rightarrow e_n$

equals some e_i for which p_i equals TRUE. If there is no such p_i, then the value of the expression is unspecified and TLC will report an error when evaluating it. For example, the value of the expression

CASE $i \in 1 \,..\, N \rightarrow$ "a" $\Box$ $i \in N \,..\, 2 * N \rightarrow$ "b"

is

- "a" if i is in $1 \,..\, (N - 1)$,
- "b" if i is in $(N + 1) \,..\, 2 * N$,
- either "a" or "b" if $i = N$,
- unspecified if i is not in $1 \,..\, 2 * N$, and TLC reports an error when evaluating it.

In the third case, whether the expression equals "a" or "b", it equals the same value every time it is evaluated.

The CASE expression

CASE $p_1 \rightarrow e_1 \;\Box\; \ldots \;\Box\; p_n \rightarrow e_n \;\Box$ OTHER e

is equivalent to

CASE $p_1 \rightarrow e_1 \;\Box\; \ldots \;\Box\; p_n \rightarrow e_n \;\Box\; \neg(p_1 \vee \ldots \vee p_n) \rightarrow e$

Thus, its value equals e if all of the p_i equal FALSE.

A LET expression allows you to make definitions local to the expression. For example,

$$\begin{array}{lll} \text{LET} & x & \triangleq \quad a + b \\ & y & \triangleq \quad a - b \\ \text{IN} & \text{IF } y > 0 \text{ THEN } x + y \text{ ELSE } x - y & \end{array}$$

equals

$$\text{IF } a - b > 0 \text{ THEN } (a + b) + (a - b) \text{ ELSE } (a + b) - (a - b)$$

Any sequence of TLA$^+$ definitions can appear between the LET and the IN. Section 5.11 on page 52 describes TLA$^+$ definitions.

5.10 Temporal Operators

A *behavior* is a nonempty sequence of states, where a state is an assignment of values to variables. A behavior of an algorithm is one that can be generated by executing the algorithm.

A *temporal formula* is a predicate on behaviors—in other words, it is true or false for any nonempty sequence of states. An algorithm satisfies a temporal formula F iff F is true of all behaviors of the algorithm.

Temporal formulas cannot appear in a $^+$CAL algorithm. They are used only in the fairness properties assumed of the algorithm's executions and in the properties asserted about the algorithm. Section 4.6 on page 35 explains how to assert fairness properties of the algorithm and how to tell TLC to check liveness properties. This section defines the TLA$^+$ temporal operators that are used to express these fairness and liveness properties. The definitions are given for infinite behaviors only. They are applied to finite behaviors by considering the behavior $\langle s_1, \ldots, s_n \rangle$ to be equivalent to the infinite behavior $\langle s_1, \ldots, s_n, s_n, s_n, \ldots \rangle$ obtained by repeating the last state forever.

5.10.1 Fairness

An *atomic operation* of an algorithm consists of all control paths that start at some label l, end at a label, and do not pass through any label. For example, this code sequence

```
L1: if x = 0 then y := y + 1
              else L2: when sem > 0 ;
                       sem := sem - 1;
    end if;
L3: ...
```

contains the atomic operations

```
L1:  if x = 0 then y := y + 1 else L2:  end if; L3:
```

and

```
L2:  when sem > 0 ; sem := sem - 1 ; L3:
```

We name an atomic operation by the label that begins it, so the second of these atomic operations is called operation `L2`.

In TLA$^+$, an *action* is a formula describing how the state changes. More precisely, it is a formula that is true or false of a pair of states. We say that $s \to t$ is an A *step* iff action A is true of the pair $\langle s, t \rangle$ of states. An action A is said to be *enabled* in a state s iff it is possible to perform an A action in state s—that is, iff there is some state t such that $s \to t$ is an A step.

For each atomic operation `L` of a $^+$CAL algorithm, the TLA$^+$ translation defines an action L that represents the operation—in other words, where $s \to t$ is an L step iff executing operation `L` in state s can produce state t. We call L an *atomic action* of the algorithm. Appendix Section B on page 57 describes how atomic operations are represented as atomic actions.

Fairness assumptions about the algorithm are expressed with the following fairness assumptions about actions:

Weak Fairness of an action A means that if it remains continuously possible to execute A, then A is eventually executed. Weak fairness of A is expressed by the temporal formula $\mathrm{WF}_{vars}(A)$ (typed `WF_vars(A)`), where *vars* is the tuple of all variables of the algorithm. This formula asserts of a behavior $\langle s_1, s_2, \ldots \rangle$ that, if there is some $i > 0$ such that A is enabled in state s_j, for all $j \geq i$, then $s_j \to s_{j+1}$ is an A step, for some $j \geq i$.

Strong Fairness of an action A means that if it is repeatedly possible to execute A, even if it is repeatedly impossible to execute A, then A is eventually executed. Strong fairness of A is expressed by the formula $\mathrm{SF}_{vars}(A)$ (typed `SF_vars(A)`). This formula asserts of a behavior $\langle s_1, s_2, \ldots \rangle$ that if A is enabled in infinitely many states s_j, then $s_j \to s_{j+1}$ is an A step, for some $j > 0$.

Strong fairness of A is stronger than (implies) weak fairness of A. In other words, if $\mathrm{SF}_{vars}(A)$ is true of a behavior σ, then $\mathrm{WF}_{vars}(A)$ is also true of σ.

As an example, let $L2$ be the atomic action corresponding to the atomic operation `L2` above (on the preceding page). Weak fairness of $L2$ means that, if the process is at control point `L2` and *sem* remains positive, then eventually operation `L2` will be executed. Strong fairness of $L2$ means that, if the process is at control point `L2` and *sem* keeps being set to a positive value, even if it keeps being reset to 0, then eventually `L2` will be executed.

A process's next-state action is defined to be the disjunction of all its atomic actions. The most common fairness assumption is weak fairness of each process's next-state action. For a $^+$CAL algorithm, weak fairness of a process's next-state action is equivalent to the conjunction of weak fairness of each of its atomic actions. Similarly, strong fairness of a process's next-state action in a $^+$CAL algorithm is equivalent to strong fairness of each of its atomic actions. An algorithm's next-state action is the disjunction of all of its atomic actions. Weak fairness of the algorithm's next-state action means that the algorithm will not halt if it is possible for some process to perform an action.

5.10.2 Liveness

The temporal properties that an algorithm should satisfy are expressed with the temporal operators "$\Box$", "$\Diamond$", and "$\leadsto$", which are defined as follows:

$\Box F$ is true of a behavior σ iff the temporal formula F is true for every suffix of σ. In other words, $\Box F$ is true for a behavior $\langle s_1, s_2, \ldots\rangle$ iff F is true on the behavior $\langle s_i, s_{i+1}, \ldots\rangle$, for all $i > 0$. Hence, if $\Box F$ is true of σ, then F is true of σ. We usually think of $\Box F$ as asserting that F is *always* true.

$\Diamond F$ is true of a behavior σ iff the temporal formula F is true of some suffix of σ. Since σ is a suffix of itself, if F is true of σ then so is $\Diamond F$. We usually think of $\Diamond F$ as asserting that F is *eventually* true.

$F \leadsto G$ asserts of a behavior σ that if τ is any suffix of σ for which F is true, then there is a suffix of τ for which G is true. In other words, $F \leadsto G$ asserts that whenever F becomes true, G must be true then or at some later point in the execution. We usually read $\leadsto$ as "leads to".

Formulas expressing liveness properties are built with these operators and state predicates. A *state predicate* is a formula that is true or false of a

state. For example, the state predicate $x > 0$ is true of those states in which the value of x is greater than 0. A state predicate is considered to be the temporal formula that is true of a behavior σ iff it is true of the first state of σ. If P is a state predicate, then $\Box P$ is true of a behavior σ iff P is true of the first state of all suffixes of the behavior—in other words, iff P is true of all states of σ. Hence, the temporal formula $\Box P$ asserts that P is an invariant of an algorithm. The formula $\Diamond P$ asserts of a behavior σ that P is true in some state of σ.

To check that you understand these temporal operators, you can verify that:

- $\Box\Diamond F$ is true of a behavior σ iff F is true for infinitely many suffixes of σ. In particular, if P is a state predicate, then $\Box\Diamond P$ asserts of σ that P is true for infinitely many states of σ.
- $\Diamond\Box F$ is true of a behavior $\langle s_1, s_2, \ldots\rangle$ iff there is some $i > 0$ such that F is true of $\langle s_j, s_{j+1}, \ldots\rangle$, for all $j \geq i$.
- $F \rightsquigarrow G$ is equivalent to $\Box(F \Rightarrow \Diamond G)$.

5.10.3 One Algorithm Implementing Another

Instead of just checking that an algorithm satisfies certain properties, you can check that it implements a complete higher-level specification describing what the algorithm is supposed to do. The TLA$^+$ book explains how to write such a specification in TLA$^+$. You can also write the specification as a more abstract $^+$CAL algorithm. In that case, you will have to show that the algorithm implements its more abstract version under an interface refinement. Interface refinement is explained in Section 10.8 of the TLA$^+$ book. In most cases, the interface refinement will be a simple data refinement.

5.11 TLA$^+$ Definitions

TLA$^+$ allows you to define operators that take zero or more arguments. Ordinary operator definitions have the two forms

$$F \triangleq expr$$
$$F(p_1, \ldots, p_n) \triangleq expr$$

(The symbol "$\triangleq$" is typed "==".) The identifier F and the formal parameters p_i must not already have a meaning. All identifiers, other than

the p_i, that appear in the expression *expr* must already have a meaning. Hence, recursive definitions cannot be written in this way. TLA$^+$ does allow recursive *function* definitions of the form

$$f[x \in S] \triangleq expr$$

For example, you can define the factorial function by

$$fact[n \in Nat] \triangleq \text{IF } n = 0 \text{ THEN } 1 \text{ ELSE } n * fact[n-1]$$

TLA$^+$ also permits definitions of binary (infix) operators. For example, the following defines $\oplus$ (typed "`(+)`") to mean addition modulo N:

$$a \oplus b \triangleq (a + b) \,\%\, N$$

Table 4 on page 64 lists all user-definable operator symbols. (Table 5 on page 65 lists the non-obvious ASCII versions of those symbols.)

Definitions of operators or functions that are used in the $^+$CAL algorithm must appear in the module before the "`BEGIN TRANSLATION`" line, or in a `define` statement as described in Section 3.6 on page 26. Except for ones in a `define` statement, definitions that use the identifiers declared or defined in the translation must come after the "`END TRANSLATION`" line. Those identifiers are listed in Section 3.8 on page 27.

TLA$^+$ does not use a semicolon or any other delimiter to end a definition. It's customary to start each definition on a new line, but that isn't necessary. Two successive definitions can be separated by any kind of space character or characters.

References

[1] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[2] Leslie Lamport. *Specifying Systems*. Addison-Wesley, Boston, 2003. Also available on the Web via a link at `http://lamport.org`.

[3] J. T. Schwartz, R. B. Dewar, E. Schonberg, and E. Dubinsky. *Programming with sets: An Introduction to SETL*. Springer-Verlag, New York, 1986.

[4] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1988.

Appendix

Section A gives the BNF grammar of $^+$CAL, and Section B describes the TLA$^+$ translation of a $^+$CAL algorithm.

A The Grammar

The algorithm must appear in a file with a TLA$^+$ module, either outside the module or within a single comment. In the latter case, it will almost surely be enclosed by "`(*`" and "`*)`". Comments within an algorithm are delimited by "`\*`" or "`(*...*)`". The grammar described here is for an algorithm with every comment removed and replaced by one or more spaces.

Here is a simplified BNF grammar for the algorithm text. It does not express the restrictions on where labels must or may not occur, which are explained in Section 3.7 on page 26. It also does not express the restrictions on what statements may occur in the body of a macro—namely, that a `while`, `call`, `return`, `goto`, or macro call may not appear there. The BNF uses the following notations.

- The square brackets "[" and "]" are BNF grouping symbols (but "`[`" and "`]`" are literals).
- $a \mid b$ means a or b.
- $a^{0,1}$ means 0 or 1 instance of a.
- a^* means 0 or more instances of a.
- a^+ means 1 or more instances of a.

Any "`;`" prescribed in the grammar may be omitted if it immediately precedes any of the following reserved words.

```
begin  do  else  elsif  end  macro  or  procedure  process
```

The grammar is:

⟨*Algorithm*⟩ ::= `--algorithm` ⟨*name*⟩
⟨ *VarDecls*⟩^{0,1}
⟨*Definitions*⟩^{0,1}
⟨*Macro*⟩*
⟨*Procedure*⟩*
[⟨*AlgorithmBody*⟩ | ⟨*Process*⟩$^+$]
`end algorithm`

⟨ *Definitions* ⟩ ::= `define` ⟨ *Defs* ⟩ `end define` [`;`]^{0,1}

⟨ *Macro* ⟩ ::= `macro` ⟨ *Name* ⟩ ([⟨ *Variable* ⟩ [`,` ⟨ *Variable* ⟩]*]^{0,1})
⟨ *AlgorithmBody* ⟩
`end macro` [`;`]^{0,1}

⟨ *Procedure* ⟩ ::= `procedure` ⟨ *Name* ⟩ ([⟨ *PVarDecl* ⟩ [`,` ⟨ *PVarDecl* ⟩]*]^{0,1})
⟨ *PVarDecls* ⟩^{0,1}
⟨ *AlgorithmBody* ⟩
`end procedure` [`;`]^{0,1}

⟨ *Process* ⟩ ::= `process` ⟨ *Name* ⟩ [`=` | `\in`] ⟨ *Expr* ⟩
⟨ *VarDecls* ⟩^{0,1}
⟨ *AlgorithmBody* ⟩
`end process` [`;`]^{0,1}

⟨ *VarDecls* ⟩ ::= [`variable` | `variables`] ⟨ *VarDecl* ⟩$^{+}$

⟨ *VarDecl* ⟩ ::= ⟨ *Variable* ⟩ [[`=` | `\in`] ⟨ *Expr* ⟩]^{0,1} `;`

⟨ *PVarDecls* ⟩ ::= [`variable` | `variables`] [⟨ *PVarDecl* ⟩ `;`]$^{+}$

⟨ *PVarDecl* ⟩ ::= ⟨ *Variable* ⟩ [`=` ⟨ *Expr* ⟩]^{0,1}

⟨ *AlgorithmBody* ⟩ ::= `begin` ⟨ *Stmt* ⟩$^{+}$

⟨ *Stmt* ⟩ ::= [⟨ *Label* ⟩ `:`]^{0,1} ⟨ *UnlabeledStmt* ⟩

⟨ *UnlabeledStmt* ⟩ ::= ⟨ *Assign* ⟩ | ⟨ *If* ⟩ | ⟨ *While* ⟩ | ⟨ *Either* ⟩ | ⟨ *With* ⟩ |
⟨ *When* ⟩ | ⟨ *Print* ⟩ | ⟨ *Assert* ⟩ | ⟨ *Skip* ⟩ | ⟨ *Return* ⟩ |
⟨ *Goto* ⟩ | ⟨ *Call* ⟩ | ⟨ *MacroCall* ⟩

⟨ *Assign* ⟩ ::= ⟨ *LHS* ⟩ `:=` ⟨ *Expr* ⟩ [`||` ⟨ *LHS* ⟩ `:=` ⟨ *Expr* ⟩]* `;`

⟨ *LHS* ⟩ ::= ⟨ *Variable* ⟩ [`[` ⟨ *Expr* ⟩ [`,` ⟨ *Expr* ⟩]* `]` | `.` ⟨ *Field* ⟩]*

⟨ *If* ⟩ ::= `if` ⟨ *Expr* ⟩ `then` ⟨ *Stmt* ⟩$^{+}$
[`elsif` ⟨ *Expr* ⟩ `then` ⟨ *Stmt* ⟩$^{+}$]*
[`else` ⟨ *Stmt* ⟩$^{+}$]^{0,1}
`end if ;`

⟨ *While* ⟩ ::= `while` ⟨ *Expr* ⟩ `do` ⟨ *Stmt* ⟩$^{+}$ `end while ;`

⟨ *Either* ⟩ ::= `either` ⟨ *Stmt* ⟩$^{+}$ [`or` ⟨ *Stmt* ⟩$^{+}$]$^{+}$ `end either ;`

⟨ *With* ⟩ ::= `with` [⟨ *Variable* ⟩ [`=` | `\in`] ⟨ *Expr* ⟩ `;`]$^{+}$
`do` ⟨ *Stmt* ⟩$^{+}$ `end with ;`

⟨ *When* ⟩ ::= `when` ⟨ *Expr* ⟩ `;`

⟨ *Print* ⟩ ::= `print` ⟨ *Expr* ⟩ `;`

⟨ *Assert* ⟩ ::= `assert` ⟨ *Expr* ⟩ `;`

⟨ *Skip* ⟩ ::= `skip ;`

⟨ *Return* ⟩ ::= `return ;`

⟨ *Goto* ⟩ ::= `goto` ⟨ *Label* ⟩ `;`

⟨ *Call* ⟩ ::= `call` ⟨ *MacroCall* ⟩

⟨ *MacroCall* ⟩ ::= ⟨ *Name* ⟩ `(` [⟨ *Expr* ⟩ [`,` ⟨ *Expr* ⟩]*]^{0,1} `)` `;`

⟨ *Variable* ⟩ ::= A TLA^{+} identifier that is not a $^{+}$CAL reserved word and is not *pc*, *stack*, or *self*.

⟨ *Field* ⟩ ::= A TLA^{+} record-component label.

⟨ *Name* ⟩ ::= A TLA^{+} identifier that is not a $^{+}$CAL reserved word.

⟨ *Label* ⟩ ::= A TLA^{+} identifier that is not a $^{+}$CAL reserved word and is not `Done` or `Error`.

⟨ *Expr* ⟩ ::= A TLA^{+} expression not containing a $^{+}$CAL reserved word or symbol.

⟨ *Defs* ⟩ ::= A sequence of TLA^{+} definitions not containing a $^{+}$CAL reserved word or symbol.

TLA^{+} expressions and definitions are described in Section 5. A TLA^{+} record-component label is any sequence of letters, digits, and "`_`" characters containing at least one non-digit and not equal to "`WF_`" or "`SF_`". A TLA^{+} identifier is a record-component that is not one of the following.

```
ASSUME  ASSUMPTION  AXIOM  CASE  CHOOSE  CONSTANT
CONSTANTS  DOMAIN  ELSE  ENABLED  EXCEPT  EXTENDS  IF  IN
INSTANCE  LET  LOCAL  MODULE  OTHER  UNION  SUBSET  THEN
THEOREM  UNCHANGED  VARIABLE  VARIABLES  WITH
```

The $^{+}$CAL reserved words are

```
assert  begin  call  do  either  else  elsif  end  goto
if  macro  or  print  procedure  process  return  skip
then  variable  variables  while  with
```

The $^{+}$CAL reserved symbols are "`:=`" and "`||`".

B The TLA$^+$ Translation

B.1 The *FastMutex* Algorithm

The TLA$^+$ translation is described with the example algorithm of *FastMutex* in Figure 2 on page 12. I have simplified the translation a bit to make it easier to read, but I have not altered its meaning. The translation begins by declaring the algorithm's variables and defining *vars* to be the tuple of all these variables.

VARIABLES x, y, b, j, pc
$vars \triangleq \langle x, y, b, j, pc \rangle$

The variable *pc* is added to describe the control state. If an algorithm contains one or more procedures, a variable *stack* is added to hold the calling stack. In addition, each formal parameter and local variable of a procedure is be declared to be a variable.

Had the algorithm contained a `define` statement, the translation would have contained two VARIABLES statements, the first declaring the variables x, y, b, and pc, and the second declaring the remaining variable j. The definitions from the `define` statement would have been put between the two VARIABLES statements.

For a multiprocess program, the translation next defines the set *ProcSet* of all process identifiers.

$ProcSet \triangleq 1 \mathinner{..} N$

Next is the definition of the initial predicate *Init* that specifies the initial values of all the declared variables. Comments indicate if the variables are global or local to a process or procedure.

$$
\begin{aligned}
Init \triangleq\ & \text{Global variables} \\
& \land x = 0 \\
& \land y = 0 \\
& \land b = [i \in 1 \mathinner{..} N \mapsto \text{FALSE}] \\
& \text{Process Proc} \\
& \land j\ \ = [self \in 1 \mathinner{..} N \mapsto \{\,\}\,] \\
& \land pc = [self \in ProcSet \mapsto \text{“ncs”}]
\end{aligned}
$$

Observe that the process-local variables and the variable *pc* are made functions with domain equal to the appropriate set of process identifiers.

Next come the action definitions. As explained in Section 5.10.1 on page 49, a TLA$^+$ action is a formula describing a pair of states—the state

before executing the action and the state after executing it. In an action, unprimed variables refer to their values before executing the action and the primed variables refer to their values after the execution.

The translation defines an action for each atomic operation of the algorithm. As explained in Section 5.10.1, an atomic operation begins at a label that is used to name the action. The first such action definition is generated by statement *ncs*. The definition is parameterized by the identifier *self*, which represents the current process's identifier. (The EXCEPT construct is explained in Section 5.7 on page 46.)

$$\begin{aligned} ncs(self) \triangleq\ & \land\ pc[self] = \text{“ncs”} \\ & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“start”}] \\ & \land\ \text{UNCHANGED } \langle x,\ y,\ b,\ j \rangle \end{aligned}$$

The conjunct $pc[self] = \text{“ncs”}$ is an *enabling condition*, meaning that the action can be executed only when it is true. It asserts that control of process *self* is at label *ncs*. The action sets $pc[self]$ to “start”; the UNCHANGED conjunct asserts that the values of all other variables are not changed. The evaluation of the `while` test does not appear explicitly in the action because it equals TRUE. The `skip` statement similarly does not appear because it has no effect.

The atomic actions corresponding to the statements labeled *start* and $l1$ are analogous.

$$\begin{aligned} start(self) \triangleq\ & \land\ pc[self] = \text{“start”} \\ & \land\ b' = [b \text{ EXCEPT } ![self] = \text{TRUE}] \\ & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“l1”}] \\ & \land\ \text{UNCHANGED } \langle x,\ y,\ j \rangle \end{aligned}$$

$$\begin{aligned} l1(self) \triangleq\ & \land\ pc[self] = \text{“l1”} \\ & \land\ x' = self \\ & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“l2”}] \\ & \land\ \text{UNCHANGED } \langle y,\ b,\ j \rangle \end{aligned}$$

Action $l2$ performs the `if` statement's test and the subsequent transfer of control.

$$\begin{aligned} l2(self) \triangleq\ & \land\ pc[self] = \text{“l2”} \\ & \land\ \text{IF } y \neq 0 \text{ THEN } pc' = [pc \text{ EXCEPT } ![self] = \text{“l3”}] \\ & \qquad\qquad \text{ELSE } pc' = [pc \text{ EXCEPT } ![self] = \text{“l5”}] \\ & \land\ \text{UNCHANGED } \langle x,\ y,\ b,\ j \rangle \end{aligned}$$

The body of the `then` clause of statement $l2$ consists of two atomic actions.

$$
\begin{array}{lll}
l3(self) & \triangleq & \land\ pc[self] = \text{“l3”} \\
 & & \land\ b' = [b \text{ EXCEPT } ![self] = \text{FALSE}] \\
 & & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“l4”}] \\
 & & \land\ \text{UNCHANGED } \langle x,\ y,\ j \rangle \\
 & & \\
l4(self) & \triangleq & \land\ pc[self] = \text{“l4”} \\
 & & \land\ y = 0 \\
 & & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“start”}] \\
 & & \land\ \text{UNCHANGED } \langle x,\ y,\ b,\ j \rangle
\end{array}
$$

The expression $y = 0$ of the `when` statement is an enabling condition of action $l4$. (Recall that this means the action can be executed only when $y = 0$ is true. The conjunct $pc[self] =$ “l4” is the other enabling condition of this action.)

Actions $l5$ and $l6$ introduce nothing new and their definitions are omitted here. Action $l7$ shows that the process's local variable j has been turned into an array indexed by $self$.

$$
\begin{array}{lll}
l7(self) & \triangleq & \land\ pc[self] = \text{“l7”} \\
 & & \land\ b' = [b \text{ EXCEPT } ![self] = \text{FALSE}] \\
 & & \land\ j' = [j \text{ EXCEPT } ![self] = 1] \\
 & & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“l8”}] \\
 & & \land\ \text{UNCHANGED } \langle x,\ y \rangle
\end{array}
$$

Action $l8$ shows how a `while` statement whose test is not identically true is translated.

$$
\begin{array}{lll}
l8(self) & \triangleq & \land\ pc[self] = \text{“l8”} \\
 & & \land\ \text{IF } j[self] \leq N \\
 & & \quad \text{THEN } \land\ \neg b[j[self]] \\
 & & \qquad\quad \land\ j' = [j \text{ EXCEPT } ![self] = j[self] + 1] \\
 & & \qquad\quad \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“l8”}] \\
 & & \quad \text{ELSE } \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“l9”}] \\
 & & \qquad\quad \land\ \text{UNCHANGED } j \\
 & & \land\ \text{UNCHANGED } \langle x,\ y,\ b \rangle
\end{array}
$$

Actions $l9$, cs, and $l11$ are obtained in a similar manner and are omitted. Actions $l10$ and $l12$ show the translation of an explicit `goto` and the transfer of control at the end of the `while` loop.

$$
\begin{array}{lll}
l10(self) & \triangleq & \land\ pc[self] = \text{“l10”} \\
 & & \land\ y = 0
\end{array}
$$

$$
\begin{array}{rcl}
 & & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“start”}] \\
 & & \land\ \text{UNCHANGED } \langle x,\ y,\ b,\ j\rangle \\
l12(self) & \triangleq & \land\ pc[self] = \text{“l12”} \\
 & & \land\ b' = [b \text{ EXCEPT } ![self] = \text{FALSE}] \\
 & & \land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“ncs”}] \\
 & & \land\ \text{UNCHANGED } \langle x,\ y,\ j\rangle
\end{array}
$$

The translation next defines $Proc(self)$ to be the next-state action of process *self* of process set *Proc*, which is the disjunction of all the atomic actions of the process. (The name of a process or process set is used only to name the process's next-state action.) A step of the process is one that satisfies formula $Proc(self)$.

$$
\begin{array}{rcl}
Proc(self) & \triangleq & ncs(self) \lor start(self) \lor l1(self) \lor l2(self) \lor l3(self) \\
 & & \quad \lor l4(self) \lor l5(self) \lor l6(self) \lor l7(self) \lor l8(self) \\
 & & \quad \lor l9(self) \lor l10(self) \lor cs(self) \lor l11(self) \\
 & & \quad \lor l12(self)
\end{array}
$$

The action *Next* is defined to be the next-state action of the entire algorithm. It is the disjunction of the next-state actions of all the processes. (The existential quantification is equivalent to the disjunction $Proc(1) \lor \ldots \lor Proc(N)$.)

$$
\begin{array}{rcl}
Next & \triangleq & \lor \exists\, self \in 1\,..\,N : Proc(self)) \\
 & & \lor\ \text{Disjunct to prevent deadlock on termination} \\
 & & \quad \land \forall\, self \in ProcSet : pc[self] = \text{“Done”} \\
 & & \quad \land\ \text{UNCHANGED } vars
\end{array}
$$

The last disjunct of *Next* is added for TLC's benefit. TLC has no way of distinguishing deadlock from termination, which is simply a desired form of deadlock. The translation therefore adds a disjunction to *Next* that allows a terminated algorithm to perform a step that does nothing. This transforms termination into an infinite no-op loop, so it is not reported as deadlock by TLC.[3] Since the *FastMutex* algorithm cannot terminate, the disjunct serves no function in this case. However, the translator is not clever enough to notice that the disjunct is unnecessary.

The translator next defines formula *Spec* to be the complete TLA specification of the algorithm. The formula will mean nothing to you if you don't know TLA, but that doesn't matter. You don't need to understand TLA to

[3] If you are familiar with TLA, then you will realize that adding this disjunct does not change the meaning of the specification, just the way TLC checks it.

use $^+$CAL. If no liveness or termination option is specified, the definition of *Spec* is

$$Spec \triangleq Init \wedge \Box[Next]_{vars}$$

With the `-wf` option specifying weak fairness of each process, the definition becomes

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge \forall self \in 1 \,..\, N : \mathrm{WF}_{vars}(Proc(self))$$

If the algorithm had procedures, then $Proc(self)$ would have been defined to be the disjunction only of the atomic actions in the process body. For each procedure P, the translation would have defined action $P(self)$ to be the disjunction of the atomic actions in the body of the procedure, and it would have conjoined to *Spec* the fairness property

$$\forall self \in ProcSet : \mathrm{WF}_{vars}(P(self))$$

Translation with the `-sf` option is the same as with the `-wf` option, except that WF is replaced everywhere by SF.

Finally, the translation defines the temporal formula *Termination* that asserts the property that the algorithm eventually terminates.

$$Termination \triangleq \Diamond(\forall self \in ProcSet : pc[self] = \text{“Done”})$$

This property is false for every execution of algorithm *FastMutex*.

B.2 Procedures

The *FastMutex* algorithm does not show how procedure calls and returns are translated. Their translation models a standard implementation using a call stack that is represented by the variable *stack*. For a multiprocess algorithm, the value of *stack* is an array of individual stacks, indexed by process identifier. You probably don't care exactly how procedure calls and returns are represented in TLA$^+$; if you do, you can just look at the translation of an algorithm containing them. However, you may need to understand how to read the value of *stack* when debugging your algorithm. This is explained with the sample algorithm of Figure 4 on the next page.

An execution of the algorithm calls procedure Q. The execution of Q calls procedure P, and that execution of P calls Q. The execution of Q following the last call prints the value of *stack* and returns. The other two procedure executions then return and the algorithm terminates.

The value of *stack* is thus printed in an execution of Q inside an execution of P inside an execution of Q. Executing the algorithm prints the following

```
--algorithm Procedures
  procedure P(pA, pB)
    variables pv = 0 ;
    begin LP1 : pv := 1 ;
                call Q("ProcP") ;
          LP2 : return ;
  end procedure ;
  procedure Q(qA)
    variables qv1 = 1; qv2 = 2;
    begin LQ1: if qA = "Mn"
                  then qv1 := 9 ;
                       call P("a", "b")
                  else print stack ;
               end if ;
          LQ2: return ;
  end procedure ;
  begin LM: call Q("Mn") ;
  end algorithm
```

Figure 4: An algorithm illustrating procedure calls.

value of *stack*. (The notation for writing records is explained in Section 5.6 on page 5.6.)

$$\begin{array}{l} \langle\, [qA \mapsto \text{“Mn”},\ qv1 \mapsto 9,\ qv2 \mapsto 2,\ pc \mapsto \text{“LP2”},\ \ \mathit{procedure} \mapsto \text{“Q”}]\,, \\ \ [pA \mapsto \{\},\ pB \mapsto \{\},\ pv \mapsto 0,\quad\ \ pc \mapsto \text{“LQ2”},\ \ \mathit{procedure} \mapsto \text{“P”}]\,, \\ \ [qA \mapsto \{\},\quad\ qv1 \mapsto 1,\ qv2 \mapsto 2,\ pc \mapsto \text{“Done”},\ \mathit{procedure} \mapsto \text{“Q”}]\ \rangle \end{array}$$

The value is a sequence of three records, one for each procedure being executed. The innermost procedure execution produced the first of these records. The *procedure* field shows that the algorithm is executing a call of Q, and the *pc* field shows that this execution will return to the statement labeled $LP2$. The remaining components show the values of the procedure's parameter qA and its local variables $qv1$ and $qv2$ when the procedure was called. The corresponding variables will be restored to those values upon the next return from procedure Q.

The second record in the sequence *stack* contains the same information for the call of procedure P. Since this was the first call of P in the call stack, the parameters pA and pB and the local variable pv contained their initial values. (Because the algorithm does not specify initial values for pA and pB, the default initial value $\{\}$ is used.)

Logic

$\land \quad \lor \quad \neg \quad \Rightarrow \quad \equiv$	
TRUE FALSE BOOLEAN	[the set {TRUE, FALSE}]
$\forall\, x \in S : p$ (1) $\quad \exists\, x \in S : p$ (1)	
CHOOSE $x \in S : p$	[An x in S satisfying p]

Sets

$= \quad \neq \quad \in \quad \notin \quad \cup \quad \cap \quad \subseteq \quad \setminus$	[set difference]
$\{e_1, \ldots, e_n\}$	[Set consisting of elements e_i]
$\{x \in S : p\}$ (2)	[Set of elements x in S satisfying p]
$\{e : x \in S\}$ (1)	[Set of elements e such that x in S]
SUBSET S	[Set of subsets of S]
UNION S	[Union of all elements of S]

Functions

$f[e]$	[Function application]
DOMAIN f	[Domain of function f]
$[x \in S \mapsto e]$ (1)	[Function f such that $f[x] = e$ for $x \in S$]
$[S \to T]$	[Set of functions f with $f[x] \in T$ for $x \in S$]
$[f$ EXCEPT $![e_1] = e_2]$ (3)	[Function $\widehat{f}$ equal to f except $\widehat{f}[e_1] = e_2$]

Records

$e.h$	[The h-field of record e]
$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$	[The record whose h_i field is e_i]
$[h_1 : S_1, \ldots, h_n : S_n]$	[Set of all records with h_i field in S_i]
$[r$ EXCEPT $!.h = e]$ (3)	[Record $\widehat{r}$ equal to r except $\widehat{r}.h = e$]

Tuples

$e[i]$	[The i^{th} component of tuple e]
$\langle e_1, \ldots, e_n \rangle$	[The n-tuple whose i^{th} component is e_i]
$S_1 \times \ldots \times S_n$	[The set of all n-tuples with i^{th} component in S_i]

(1) $x \in S$ may be replaced by a comma-separated list of items $v \in S$, where v is either a comma-separated list or a tuple of identifiers.

(2) x may be an identifier or tuple of identifiers.

(3) $![e_1]$ or $!.h$ may be replaced by a comma separated list of items $!\,a_1 \cdots a_n$, where each a_i is $[e_i]$ or $.h_i$.

Table 1: The operators of TLA$^+$.

IF p THEN e_1 ELSE e_2	[e_1 if p true, else e_2]
CASE $p_1 \to e_1 \,\Box\, \ldots \,\Box\, p_n \to e_n$	[Some e_i such that p_i true]
CASE $p_1 \to e_1 \,\Box\, \ldots \,\Box\, p_n \to e_n \,\Box$ OTHER $\to e$	[Some e_i such that p_i true, or e if all p_i are false]
LET $d_1 \triangleq e_1 \;\ldots\; d_n \triangleq e_n$ IN e	[e in the context of the definitions]

$$\begin{array}{ll} \land\ p_1 & \text{[the conjunction } p_1 \land \ldots \land p_n\text{]} \\ \quad\vdots & \\ \land\ p_n & \end{array} \qquad \begin{array}{ll} \lor\ p_1 & \text{[the disjunction } p_1 \lor \ldots \lor p_n\text{]} \\ \quad\vdots & \\ \lor\ p_n & \end{array}$$

Table 2: Miscellaneous constructs.

$\Box F$	[F is always true]
$\Diamond F$	[F is eventually true]
$\mathrm{WF}_e(A)$	[Weak fairness for action A]
$\mathrm{SF}_e(A)$	[Strong fairness for action A]
$F \rightsquigarrow G$	[F leads to G]

Table 3: Temporal operators.

$+$ (1)	$-$ (1)	$*$ (1)	$/$ (2)	$\circ$ (3)	++
$\div$ (1)	$\%$ (1)	$\hat{}$ (1,4)	$..$ (1)	$\ldots$	$--$
$\oplus$ (5)	$\ominus$ (5)	$\otimes$	$\oslash$	$\odot$	**
$<$ (1)	$>$ (1)	$\leq$ (1)	$\geq$ (1)	$\sqcap$	//
$\prec$	$\succ$	$\preceq$	$\succeq$	$\sqcup$	$\hat{}\,\hat{}$
$\ll$	$\gg$	$<:$	$:>$ (6)	&	&&
$\sqsubset$	$\sqsupset$	$\sqsubseteq$ (5)	$\sqsupseteq$	$\mid$	%%
$\subset$	$\supset$		$\supseteq$	$\star$	@@ (6)
$\vdash$	$\dashv$	$\models$	$=\!\mid$	$\bullet$	##
$\sim$	$\simeq$	$\approx$	$\cong$	\$	\$\$
$\bigcirc$	::=	$\asymp$	$\doteq$	??	!!
$\propto$	$\wr$	$\uplus$			

(1) Defined by the *Naturals*, *Integers*, and *Reals* modules.
(2) Defined by the *Reals* module.
(3) Defined by the *Sequences* module.
(4) $x \hat{} y$ is printed as x^y.
(5) Defined by the *Bags* module.
(6) Defined by the *TLC* module.

Table 4: User-definable operator symbols.

$\land$	`/\` or `\land`	$\lor$	`\/` or `\lor`	$\Rightarrow$	`=>`
$\neg$	`~` or `\lnot` or `\neg`	$\equiv$	`<=>` or `\equiv`	$\triangleq$	`==`
$\in$	`\in`	$\notin$	`\notin`	$\neq$	`#` or `/=`
$\langle$	`<<`	$\rangle$	`>>`	$\Box$	`[]`
$<$	`<`	$>$	`>`	$\Diamond$	`<>`
$\leq$	`\leq` or `=<` or `<=`	$\geq$	`\geq` or `>=`	$\rightsquigarrow$	`~>`
$\ll$	`\ll`	$\gg$	`\gg`	$\stackrel{+}{\rightarrow}$	`-+->`
$\prec$	`\prec`	$\succ$	`\succ`	$\mapsto$	`|->`
$\preceq$	`\preceq`	$\succeq$	`\succeq`	$\div$	`\div`
$\subseteq$	`\subseteq`	$\supseteq$	`\supseteq`	$\cdot$	`\cdot`
$\subset$	`\subset`	$\supset$	`\supset`	$\circ$	`\o` or `\circ`
$\sqsubset$	`\sqsubset`	$\sqsupset$	`\sqsupset`	$\bullet$	`\bullet`
$\sqsubseteq$	`\sqsubseteq`	$\sqsupseteq$	`\sqsupseteq`	$\star$	`\star`
$\vdash$	`|-`	$\dashv$	`-|`	$\bigcirc$	`\bigcirc`
$\models$	`|=`	$=\!\!\mid$	`=|`	$\sim$	`\sim`
$\rightarrow$	`->`	$\leftarrow$	`<-`	$\simeq$	`\simeq`
$\cap$	`\cap` or `\intersect`	$\cup$	`\cup` or `\union`	$\asymp$	`\asymp`
$\sqcap$	`\sqcap`	$\sqcup$	`\sqcup`	$\approx$	`\approx`
$\oplus$	`(+)` or `\oplus`	$\uplus$	`\uplus`	$\cong$	`\cong`
$\ominus$	`(-)` or `\ominus`	$\times$	`\X` or `\times`	$\doteq$	`\doteq`
$\odot$	`(.)` or `\odot`	$\wr$	`\wr`	x^y	`x^y` (2)
$\otimes$	`(\X)` or `\otimes`	$\propto$	`\propto`	x^+	`x^+` (2)
$\oslash$	`(/)` or `\oslash`	“s”	`"s"` (1)	x^*	`x^*` (2)
$\exists$	`\E`	$\forall$	`\A`	$X^\#$	`x^#` (2)
$\mathbf{\exists}$	`\EE`	$\mathbf{\forall}$	`\AA`	$'$	`'`
$]_v$	`]_`v	$\rangle_v$	`>>_`v		
WF_v	`WF_`v	SF_v	`SF_`v		
┌────	`--------` (3)	────┐	`--------` (3)		
├────	`--------` (3)	└────	`========` (3)		

(1) s is a sequence of characters.
(2) x and y are any expressions.
(3) a sequence of four or more `-` or `=` characters.

Table 5: The ASCII representations of typeset symbols.

Index

" (double quote), 41
^ (exponentiation), 41
|| (+CAL separator), 4
↦ (TLA+ constructor), 10, 46
\ (set difference), 44
* (end-of-line comment), 5
− (minus), 41
¬ (negation), 42
→ (TLA+ operator), 45
⇝ (leads to), 51
÷ (integer division), 41
+ (plus), 41
+CAL, 2
  expression, 17, 41–49
  grammar, 54–56
× (Cartesian product), 47
⇒ (implies), 42
≡ (equivalence), 42
≜ (defined to equal), 26, 52
.. (integer interval), 6, 41
∗ (multiplication), 41
∘ (sequence concatenation), 48
  for strings, 42
□ (always), 51
◇ (eventually), 51
< (less than), 41
<- (TLC replacement), 32
≤ (less than or equal), 41
⊆ (subset), 44
(∗ . . . ∗) (comment), 5
$f[e]$ (function application), 10, 45
$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ (record constructor), 46
$[x \in S \mapsto e]$ (function constructor), 10, 45
$[h_1 : S_1, \ldots, h_n : S_n]$ (set of records), 46
$\langle e_1, \ldots, e_n \rangle$ (tuple/sequence), 47
{ } (empty set), 35, 44
$\{e_1, \ldots, e_n\}$ (set), 43
$\{x \in S : P(x)\}$ (set constructor), 44
$\{e(x) : x \in S\}$ (set constructor), 44
> (greater than), 41
≥ (greater than or equal), 41
∧ (conjunction), 42
∩ (set intersection), 44
∨ (disjunction), 42
∪ (set union), 44
:> (TLA+ operator), 34, 45
! (in EXCEPT), 46, 47
% (modulus), 9, 41
@@ (TLA+ operator), 34, 45
∀ (universal quantification), 43
∃ (existential quantification), 43
ε, Hilbert's, 45
∈ (set membership), 44
  in `with` statement, 20
  in variable initialization, 5

action, 50, 57
  atomic, 50
  corresponding to label, 28
  defined for procedure, 28
  defined for process set, 28
  defined for single process, 28
  enabled, 50
  next-state, 28, 51, 60
algorithm
  beginning of, 5
  language, 2
  multiprocess, 10
  placement of, 6
  uniprocess, 10
  versus program, 2

always (□), 51
and (∧), 42
angle brackets (⟨ ⟩), 47
Append (TLA$^+$ operator), 48
array, 45
ASCII representation of symbol, 65
`assert` (+CAL statement), 9, 14, 21
assertion, failure of, 21
assignment, 17–18
  in `CONSTANT` statement, 32
  multiple, 4, 18
  to a component, 17
assumption, fairness, 36
atomic action, 50
atomic operation, 49
  translation of, 58

`BEGIN TRANSLATION`, 6
behavior, 49
binary operator, defining, 53
BNF grammar, 54
body of process, 22
book, TLA$^+$, 1
Boolean
  operator, 42
  value, 42
brackets, square, 10
  in BNF grammar, 54
built-in operator of TLA$^+$, 6
bulleted list, 42

`call` (+CAL statement), 23
  as assignment, 24
  not allowed in macro body, 25
  translation, 61
Cardinality (TLA$^+$ operator), 45
CASE (TLA$^+$ expression), 48
cfg file, *see* configuration file
checking
  assertion, 9
  invariant, 14, 35
  multiple invariants, 14
  termination, 9
  the algorithm, 30
CHOOSE (TLA$^+$ operator), 45
comment
  in +CAL algorithm, 5, 54
  in configuration file, 7
  nested, 5
comparable values, 22
component, assignment to, 17
computational complexity, 2
configuration file, 7
  rewriting, 30
conjunction (∧), 42
conjunctions, bulleted list of, 42
CONSTANT (TLA$^+$ statement), 6, 31
constant
  operator, 63
  parameters, 31
`CONSTANT` (TLC cfg file statement), 7, 31, 32
constraint, 33
`CONSTRAINT` (TLC cfg file statement), 33
constructor, set, 44
control path, 16

data refinement, 52
deadlock, 38, 60
`-deadlock` (TLC option), 38
debugging, 34
declaration, operator, 32
`define` (+CAL statement), 26
  translation of, 57
definition, 52
  of binary operator, 53
  placement of, 14, 26, 53
`-depth` (TLC option), 7
difference, set (\), 44

disjunction ($\vee$), 42
disjunctions, bulleted list of, 42
division, integer ($\div$), 41
`do` (clause of `while` statement), 20
DOMAIN (TLA$^+$ operator), 45
domain of a function, 45
`Done` (implicit label), 5
    not usable as label, 27

`either` ($^+$CAL statement), 19
    when label must follow, 19, 27
`else` (clause of `if` statement), 18
`elsif` (clause of `if` statement), 19
empty set, 35, 44
enabled, 50
enabling condition, 58
end of module, 6
`END TRANSLATION`, 6
equivalence ($\equiv$), 42
error
    found by TLC, 34
    found by translator, 30
    message, 10
    multiply-defined symbol, 17, 27, 29
    syntactic, 33
`Error` (implicit label), 23
    not usable as label, 27
Euclid's algorithm, 4
eventually ($\Diamond$), 51
EXCEPT (TLA$^+$ construct), 46–47
    used in translation, 33, 58
exclusion, mutual, 13
execution of multiprocess algorithm, 23
existential quantification, 43
exponentiation, 41
expression
    $^+$CAL, 17, 41–49
    changed by translator, 33
    not parsed by translator, 31
    simple, 32
EXTENDS (TLA$^+$ statement), 6

factorial, definition of, 53
failure of assertion, 21
fairness, 36, 50
FALSE, 42
Fast Mutual Exclusion Algorithm, 10
FastMutex ($^+$CAL algorithm), 12, 57
field of record, 46
file name, 6, 30
file, cfg, *see* configuration file
FiniteSets module, 45
formal parameters of a procedure, 23
    initializing, 35
formula, temporal, 49
function, 45
    domain of, 45
    of multiple arguments, 46
    recursive definition of, 53

gcd, 4, 9
Goldbach's conjecture, 3
Gonthier, Georges, 1
`goto` ($^+$CAL statement), 22
    not allowed in macro body, 25
grammar of $^+$CAL, 54–56
greatest common divisor, 4, 9

Head (TLA$^+$ operator), 47
`-help` (translator option), 30
Hilbert's ε, 45

identifier
    defined by translation, 27
    process, 22
    reassigning meaning of, 17, 25, 27, 28
    renamed by translation, 28
`if` ($^+$CAL statement), 18

when label must follow, 19, 27
IF (TLA$^+$ expression), 48
iff (if and only if), 9
implementing a specification, 52
implementing an algorithm, 52
implication ($\Rightarrow$), 42
imported module, 6
indentation used to eliminate parentheses, 43
infix operator, 64
defining, 53
Init (defined by translation), 28, 57
initial predicate, 28, 57
initializing
procedure parameter, 35
procedure variable, 23
variable, 5, 11, 35
for checking termination, 10
Int (set of integers), 41
integer division ($\div$), 41
Integers module, 41
interface refinement, 52
intersection ($\cap$), 44
invariant, 14
checking, 14, 35
`INVARIANT` (TLC cfg file statement), 14

Java, 7

keywords, 56

label, 4, 56
action corresponding to, 28
after
`call` statement, 24, 27
`either` statement, 19, 27
`if` statement, 19, 27
error caused by omitting, 31
may not appear in macro body, 25, 27
must follow
`goto` statement, 22, 27
`return` statement, 24, 27
naming an atomic operation, 50
needed at beginning of
algorithm body, 5, 26
procedure body, 23, 26
process body, 23, 26
not permitted in `with` statement, 21, 27
of `while` statement, 20, 26
rules for, 26
language, algorithm, 2
language, programming, 2
leads to ($\rightsquigarrow$), 51
Len (TLA$^+$ operator), 48
for strings, 42
list, 47
liveness, 35, 51
checking and symmetry, 40
local variable
declaration, 22
of procedure, 23
of process, 11

macro, 25, 54
call, 25
not allowed in macro body, 25
parameter, 25
`macro` ($^+$CAL statement), 25
Marzullo, Keith, 1
meaning, reassigning in TLA$^+$, 17, 25, 27, 28
membership, set, 44
mode,
model-checking, 8
simulation, 7
model checker, 7
model value, 32
model-checking mode, 8

module
end of, 6
imported, 6
name, 6
Naturals, 6
TLA$^+$, 6
TLC, 6
modulus (%), 41
multiple arguments, function of, 46
multiple assignment, 4, 18, 31
multiply-defined symbol (parsing error), 17, 27, 29
multiprocess algorithm, 10
execution of, 23
multiprocessor computer, running TLC on, 39
mutual exclusion, 13

name of
file, 6, 30
module, 6
process, 22
process set, 22, 60
Nat (set of natural numbers), 41
natural numbers, 6
Naturals module, 6, 41
negation (¬), 42
nested comments, 5
nested quantifiers, 43
Next (defined by translation), 28, 60
next-state action
of algorithm, 28, 60
of process, 51
`-nocfg` (TLC option), 30
`-nof` (translator option), 37
nondeterminism
in `either` statement, 19
in `with` statement, 20
not expressed with CHOOSE, 45
not (¬), 42
`null` (TLC error message), 34
numbers, 41
natural, 6

objects, 3
operation, atomic, 49
operator
Boolean, 42
constant, 63
declaration, 32
infix, 64
temporal, 64
TLA$^+$, 41–52
user-definable, 64
or (∨), 42
`or` (clause of `either` statement), 19
output of TLC, 33

parameter
as variable, 10
assigning value to, 7
constant, 31
declaration of, 6, 31
of macro, 25
of procedure, 23
initializing, 35
parentheses, eliminated by indentation, 43
parser, 33–34
path, control, 16
pc (variable), 14, 27, 57
in `define` statement, 26
in expressions, 14
value of, 14
Permutations (TLA$^+$ operator), 39
power set, 44
predicate, initial, 28, 57
predicate, state, 51
`print` ($^+$CAL statement), 5, 8, 21
debugging with, 34

Print (TLA$^+$ operator), 34
procedure, 23
  action defined for, 28
  call, 23
  parameter, 23
    as variable, 10
    initializing, 35
  return from, 24
  returning a value, 24
  translation of, 61–62
`procedure` (+CAL construct), 23
process, 22
  action defined for, 28
  body of, 22
  identifier of, 22
  individual, 22
  name of, 22
  next-state action of, 51
  set, 22
    action defined for, 28
    name of, 22, 60
  variable, 11
`process` (+CAL construct), 11, 22
ProcSet (defined by translation), 27, 57
program versus algorithm, 2
programming language, 2
`PROPERTY` (TLC cfg file statement), 36

quantification, 43
quantifier nesting, 43

reachable states, 8, 33
record, 46
  as function, 46
recursion, tail, 24
recursive definition, 53
refinement, data, 52
refinement, interface, 52
renaming by translation, 28
replacement in `CONSTANT` statement, 32
reserved words, 56
`return` (+CAL statement), 24
  as assignment, 24
  not allowed in macro body, 25
  translation, 61
rules for labels, 26

SANY parser, 33–34
Sedgewick, Robert, 4
`self` (+CAL identifier), 11, 23, 24, 58
semicolon (`;`)
  not used in TLA$^+$ definitions, 26, 53
  omitting, 16, 54
Seq (TLA$^+$ operator), 47
sequence, 47–48
  of statements, 16
  of states printed by TLC, 34
Sequences Module, 47
set, 43–45
  constructor, 44
  difference (\), 44
  membership, 44
  of all . . . , 44
  power, 44
  process, 22
SETL, 2
SF (TLA$^+$ operator), 36
`-sf` (translator option), 37, 61
simple expression, 32
`-simulate` (TLC option), 7
simulation mode, 7
`skip` (+CAL statement), 13, 21
Spec (defined by translation), 28, 60
special character in string, 41
`SPECIFICATION` (TLC cfg file statement), 38

specification of the algorithm, 60
square brackets, 10
    in BNF grammar, 54
stack (variable), 27, 57, 61
    in `define` statement, 26
    value, 61
state, 34
    predicate, 51
    reachable, 8, 33
    sequence printed by TLC, 34
statement sequence, 16
step, 16, 50
    of a +CAL algorithm, 4
    of an algorithm, 2
string, 41–42
    treatment by TLC, 41
strong fairness, 36, 50
struct, 46
subset, 44
SUBSET (TLA+ operator), 44
substitution in TLC, 32
symbol
    symbol, ASCII representation, 65
    typing, 4
    user-definable, 64
`SYMMETRY` (TLC cfg file statement), 39
symmetry used by TLC, 39–40
syntactic error, 33

Tail (TLA+ operator), 47
tail recursion, 24
temporal
    formula, 49
    operator, 49–52, 64
termination, 35, 60
    checking, 9
Termination (property defined by translation), 9, 61
`-termination` (translator option), 36
`then` (clause of `if` statement), 19
threads, running TLC with multiple, 39
TLA+, 2
    book, 1
    definition, 52
        placement of, 14, 17, 53
    module, putting algorithm in, 6
    operator, 41–52
    tools web page, 30
TLC model checker, 7, 30
    error found by, 34
    error message, 10
    on multiprocessor computer, 39
    output of, 33
    running, 7
    treatment of strings, 41
    use of symmetry, 39
TLC module, 6, 34, 39
    needed for `assert` statement, 21
    needed for `print` statement, 21
translation, 57–62
    identifiers defined by, 27
    identifiers renamed by, 28
    of atomic operation, 58
    of procedure, 61–62
    variables declared by, 27
translator, 6
    expressions changed by, 33
    running the, 7, 30–31
TRUE, 42
tuple, 47
type correctness, 35
types, 3
typing symbols, 4

unary minus, 41
UNCHANGED (TLA+ operator), 58
union (∪), 44
UNION (TLA+ operator), 44

uniprocess algorithm, 10
universal quantification, 43
user-definable symbol, 64

value
    Boolean, 42
    model, 32
    returned by procedure, 24
values, comparable, 22
variable, 56
    declared by translation, 10, 27
    initializing, 5, 11, 23, 35
      for checking termination, 10
    local, declaration of, 22
    multiple assignments to, 18, 24, 27
    of procedure, 23
    of process, 11
`variable` (+CAL statement), 11
vars (defined by translation), 28, 57

weak fairness, 36, 50
web page, TLA+ tools, 30
WF (TLA+ operator), 36
`-wf` (translator option), 36, 61
`-wfNext` (translator option), 37
`when` (+CAL statement), 13, 20
`while` (+CAL statement), 20
    not allowed in macro body, 25
`with` (+CAL statement), 13, 20
`-workers` (TLC option), 39